The Dissertation Committee for Jing-Tang Keith Jang
certifies that this is the approved version of the following dissertation:

The Size and Depth of Boolean Circuits

Committee:

Anna Gál, Supervisor

Adnan Aziz

Greg Plaxton

Vijaya Ramachandran

David Zuckerman

The Size and Depth of Boolean Circuits

by

Jing-Tang Keith Jang, BS; MSCS

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2013

To my family

Acknowledgments

I would like to thank my advisor, Anna Gál, for the time she has spent helping me shaping my research. Most of the results in this dissertation is joint work with her. I would also like to thank my committee members for invaluable discussions.

This research was partially supported by an MCD fellowship from Department of Computer Science, University of Texas at Austin, and NSF Grant CCF-1018060.

I would like to thank my family in Taiwan for their confidence in me since I was a kid. I would also like to thank my wife's family for their kind support over the years.

I would like to thank my children, Ariel and Jonathan, for their determination to improve my time-management skills ever since they were born. They have taught me much more than I can teach them. Finally, I would like to thank my wife, Hsiao-Yen Kuo, for her continuous support over the years. This dissertation would not be possible without her.

Jing-Tang Keith Jang

The University of Texas at Austin
August 2013

The Size and Depth of Boolean Circuits

Jing-Tang Keith Jang, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Anna Gál

We study the relationship between size and depth for Boolean circuits. Over four decades, very few results were obtained for either special or general Boolean circuits. Spira [129] showed in 1971 that any Boolean formula of size s can be simulated in depth $O(\log s)$. Spira's result means that an arbitrary Boolean expression can be replaced by an equivalent "balanced" expression, that can be evaluated very efficiently in parallel. For general Boolean circuits, the strongest known result [101, 34] is that Boolean circuits of size s can be simulated in depth $O(s/\log s)$.

We obtain significant improvements over the general bounds for the size versus depth problem for special classes of Boolean circuits. We show that every layered Boolean circuit of size s can be simulated by a layered Boolean circuit of depth $O(\sqrt{s \log s})$. For planar circuits and synchronous circuits of size s, we obtain simulations of depth $O(\sqrt{s})$. Improving any of the above results by polylog factors would immediately improve the bounds for general circuits.

We generalize Spira's theorem and show that any Boolean circuit of size s with segregators of size $f(s)$ can be simulated in depth $O(f(s)\log s)$. This improves and generalizes a simulation of polynomial-size Boolean circuits of constant treewidth k in depth $O(k^2 \log n)$ by Jansen and Sarma [63]. Since the existence of small balanced separators in a directed acyclic graph implies that the graph also has small segregators, our results also apply to circuits with small separators. Our results imply that the class of languages computed by non-uniform families of polynomial size circuits that have constant size segregators equals non-uniform NC^1.

As an application of our simulation of circuits in small depth, we show that the Boolean Circuit Value problem for circuits with constant size segregators (or separators) is in deterministic $SPACE(\log^2 n)$. Our results also imply that the Planar Circuit Value problem, which is known to be P-Complete [53], is in $SPACE(\sqrt{n}\log n)$. We also show that the Layered Circuit Value and Synchronous Circuit Value problems, which are both P-complete [54], are in $SPACE(\sqrt{n})$.

Our study of circuits with small separators and segregators led us to obtain space efficient algorithms for computing balanced graph separators. We extend this approach to obtain space efficient approximation algorithms for the search and optimization versions of the SUBSET SUM problem, which is one of the most studied NP-complete problems.

Finally we study the relationship between simultaneous time and space bounds on Turing machines and Boolean circuit depth. We observe a new connection between planar circuit size and simultaneous time and space products of input-oblivious Turing machines. We use this to prove quadratic lower bounds on the product of time and space for several explicit functions for input-oblivious Turing machines.

Contents

List of Figures

Chapter 1

Introduction

1.1 Basic Definitions

1.1.1 Turing Machines

For the definition of Turing machines, we follow the convention of considering Turing machines with a separate *read-only* input tape, and additional work tapes. If the machine has to produce an output string (instead of just accepting or rejecting its input), then we also assume a separate *write-only* output tape. The space used by a Turing machine on a given input is defined as the number of work tape cells visited during the computation over all work tapes. The input tape and the output tape do not contribute to the space bound of the computation. This allows us to consider computations with sublinear space.

$DTIME(t(n))$ denotes the classes of languages decidable by deterministic Turing machines with a read-only input tape and a constant number of work tapes in time $O(t(n))$. $SPACE(s(n))$ denotes the class of languages decidable by deterministic Turing machines with a separate read-only input tape using $O(s(n))$ space on the work tapes.

In the following, whenever we talk about space bounds of Turing Machines, it is assumed that the input tape is read-only, the output tape (if any) is write-only and the space bound refers to the total space used on the work tapes. See Papadimitriou [100] for more details on space bounded Turing machines.

1.1.2 Boolean Circuits

A Boolean circuit is a labeled directed acyclic graph (DAG), where every node is labeled by either a variable from $\{x_1, \ldots, x_n\}$, or a Boolean operation. The set of available operations we are allowed to use is called the *basis* of the circuit. A given basis B is called *complete* if any Boolean function can be computed by a circuit using only operations from B. The *inputs* of a Boolean circuit are the nodes with in-degree (fanin) zero, and the *outputs* of a Boolean circuit are the nodes with out-degree (fanout) zero. The inputs are labeled by variables $x_1, \ldots, x_n$. A Boolean circuit may have multiple outputs. We refer to the nodes with non-zero in-degree as *gates*. We will typically consider Boolean circuits with gates of fanin at most 2 from the basis $\{\wedge, \vee, \neg\}$ (unless stated otherwise). The gates are labeled by operations from the basis. A formula (or tree-like circuit) is a circuit whose fanout is one for every gate except the output.

Definition 1.1.1. The size of a Boolean circuit is the number of its gates. The *depth of a gate* g is the length of the longest path from any input to g. For circuits with one output, the *depth of a circuit* C is the depth of the output gate. For circuits with multiple outputs, the *depth of a circuit* C is the depth of the output gate with the largest depth.

Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, we write $C(f)$ and $D(f)$ to denote the smallest size and depth of any circuit over $\{\vee, \wedge, \neg\}$ computing f, respectively. Note that $C(f)$ and $D(f)$ are not necessarily achieved by the same circuit. See [146] for more on Boolean circuits.

Given a DAG G, a *subDAG* of G is a subgraph of G. The *source* of G are the nodes with no immediate predecessors, and the *sink* of G are the nodes with no immediate successors. We say that a subDAG is *weakly-connected* if the underlying undirected graph is connected. Given a Boolean circuit C and a gate g in C, the *subcircuit* at g is a weakly-connected subDAG of C with g as output. So a subDAG may not be weakly-connected, since it might consists of several disjoint weakly-connected subDAGs. On the other hand, a subcircuit must be weakly-connected. The inputs of the subcircuit could be any nodes in C upon which g depends on topologically. A circuit is *planar* if we can find an embedding for the circuit on the two-dimensional plane such that no two edges cross. *Semi-unbounded fanin* Boolean

(arithmetic) circuits are those circuits with constant fanin $\wedge$ (resp. $\times$) gates and unbounded fanin $\vee$ (resp. $+$) gates.

1.1.3 Uniform Circuits

Before we define the uniformity of circuits, we need to define the description of a circuit. There are several common ways to define the description of a circuit. To be specific, we use the following definition.

Definition 1.1.2. The *description* of a Boolean circuit is a sequence of circuit inputs $x_1, \ldots, x_n$ and the following quadruples:

$$\langle name, type, child1, child2 \rangle,$$

where *name* is the name of a gate, *type* is one of $\wedge$, $\vee$, or $\neg$, and *child*1 and *child*2 are the inputs to the gate. *child*2 can be empty for gates of type $\neg$.

Note that *child*1 and *child*2 can be either gates or circuit inputs.

Definition 1.1.3. A family of Boolean circuits $\{C_n\}$ is called *$h(n)$-space uniform*, if there exists a deterministic Turing machine M that on input 1^n, outputs the description of C_n using space $O(h(n))$ for all n. In particular, $\{C_n\}$ is *logspace uniform* if $h(n) = \log n$.

1.2 The Problem and Its History

In this dissertation, we study the relationship between the size and depth of Boolean circuits. We consider general Boolean circuits and special classes of circuits. Pippenger and Fischer [103] showed that for $t(n) \geq n$, $DTIME(t(n))$ can be simulated by logspace uniform families of circuits of size $O(t(n) \log t(n))$. Borodin [11] showed the following theorem.

Theorem 1.2.1. *[11] For $s(n) \geq \log n$, languages computed by $s(n)$-space uniform families of circuits of depth $s(n)$ are contained in $SPACE(s(n))$. Furthermore, any language decided in simultaneous time $t(n)$ and space $s(n)$ can be decided by $s(n)$-space uniform families of circuits of depth $O(s(n) \log t(n)) = O(s^2(n))$.*

It is also known that circuit depth is related to parallel computation time [130]. These results show that the study of circuit size versus depth helps to investigate the relationship between sequential and parallel computation time, as well as time versus space in sequential computation. However, very little is known about the size versus depth question for general Boolean circuits. For general Boolean circuits, the best known result so far is the following theorem, which was first proved by Paterson and Valiant [101]. Dymond and Tompa [34] later gave another proof of this result using a different method.

Theorem 1.2.2. *[101, 34] Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, we have $D(f) = O(C(f)/\log C(f))$.*

On the other hand, it can be easily shown that $D(f) = \Omega(\log C(f))$. Theorem 1.2.2 leaves a huge gap ($\log C(f)$ versus $C(f)/\log C(f)$) for circuits of any size. McColl and Paterson [95] showed that every Boolean function depending on n variables has circuit depth at most $n + 1$. There is an even stronger result by Gaskov [48] showing that circuit depth is at most $n - \log\log n + 2 + o(1)$. This gives a much stronger bound on depth than Theorem 1.2.2 for functions that require circuits of large size. In particular, for $f : \{0,1\}^n \to \{0,1\}$ such that $C(f)$ is exponential in n, [95] and [48] give essentially tight bounds on depth. However, for functions that can be computed by subexponential-size circuits, there is still a large gap. Note that Theorem 1.2.2 gives a stronger result than [95] and [48] only when $C(f) = o(n \log n)$. Improving Theorem 1.2.2 would yield improvements over [95] and [48] for larger $C(f)$ as well.

For general Boolean circuits, the simulating depth $O(t(n)/\log t(n))$ in Theorem 1.2.2 is very close to the circuit size. On the other extreme, consider tree-like circuits, where every gate has fanout at most 1. Note that tree-like circuits are also referred to as *formulas* in circuit complexity, and we will use both terms in the followings. For tree-like circuits, Spira [129] proved the following theorem.

Theorem 1.2.3. *[129] Let F be any Boolean formula of size s. Then F can be simulated by an equivalent tree-like circuit of depth $O(\log s)$.*

There are several results improving or extending Spira's theorem. Bonet and Buss [10] improved the constants in the depth bounds and the size of the simulation for Boolean formulas. Wegener [145] proved the statement for monotone

Boolean formulas, that any monotone Boolean formula of size s can be simulated by a *monotone* Boolean formula of depth $O(\log s)$. Brent [15], Bshouty, Cleve, and Eberly [16] extended Spira's theorem to arithmetic formulas. All these results study formulas, i.e. tree-like circuits with fanout 1.

1.3 Related Questions

There are many important questions related to Theorem 1.2.2, even for the case of special classes of circuits. Notice that we are not only interested in improving the depth of the simulating circuit, but also if the simulating circuit can be generated space-efficiently.

1.3.1 Time versus Space

Hopcroft, Paul, and Valiant [59] proved the following analogous theorem about sequential time and space, and Adleman and Loui [2] later gave an alternative proof.

Theorem 1.3.1. *[59, 2]* $DTIME(t(n)) \subseteq SPACE(t(n)/\log t(n))$.

By the results of [103] and [11], improving Theorem 1.2.2 by at least a polylog factor in the logspace uniform setting immediately improves Theorem 1.3.1. This also motivates the search for space-efficient algorithms to generate the simulating circuit families.

1.3.2 Time versus Depth

We first define alternating Turing machines as follows.

Definition 1.3.2. [100] An *alternating Turing machine* is a non-deterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states K is partitioned into two sets K_{AND} and K_{OR}. Let x be an input, and consider the tree of computations of N on x. Each node in this tree is a configuration of the machine, and includes the step number of the machine. Define now recursively, starting from the leaves of the trees and going up, a subset of these configurations, called the *eventually accepting configurations*, as follows: First, all leaf configurations with state "yes" are eventually accepting. A configuration C with state in K_{AND} is eventually accepting iff all

of its immediate successor configurations are eventually accepting. A configuration C with state in K_{AND} is eventually accepting iff at least one of its immediate successor configurations are eventually accepting. Finally, we say that N *accepts* x if the initial configuration is eventually accepting. We say that an alternating machine N *decides* a language L if N accepts all strings $x \in L$ and rejects all strings $x \notin L$.

We write $ATIME(t(n))$ to denote the class of all languages decided by an alternating Turing machine, where all computations on input x halt after at most $t(|x|)$ steps.

Dymond and Tompa [34] showed the following theorem.

Theorem 1.3.3. *[34]* $DTIME(t(n)) \subseteq ATIME(t(n)/\log t(n))$.

Ruzzo [118] showed the following theorem.

Theorem 1.3.4. *[118] Any language in* $ATIME(t(n))$ *can be decided by a uniform Boolean circuit family of depth* $O(t(n))$.

An immediate corollary is that any language in $DTIME(t(n))$ can be decided by a uniform Boolean circuit family of depth $O(t(n)/\log t(n))$. Pippenger and Fischer [103] showed that for $t(n) \geq n$, $DTIME(t(n))$ can be simulated by logspace uniform families of circuits of size $O(t(n)\log t(n))$. Thus improving Theorem 1.2.2 by an order strictly greater than $\log^2 C(f)$ in the logspace uniform setting immediately improves the simulation of deterministic time by circuit depth.

1.3.3 Parallel Computation Time

The PRAM is a parallel computation model first defined by Fortune and Wiley [42]. A *PRAM* consists of a set of one-processor RAMs (cf. [100]), called *processors*, plus an infinite number of *global memory*. An instruction of a processor can access its own memory or the global memory. We classify PRAMs according to restrictions on global memory access. An *EREW* (Exclusive-Read and Exclusive-Write) PRAM is a PRAM where simultaneous access to any global memory location is not allowed for both reads and writes. A *CREW* (Concurrent-Read and Exclusive-Write) PRAM allows simultaneous reads but not writes. A *CRCW* (Concurrent-Read and Concurrent-Write) PRAM allows simultaneous reads and writes. We can further classify the CRCW model according to the methods resolving simultaneous writes.

The *CRCW Priority* model is a CRCW where there is a linear ordering on the processors, and the minimum numbered processor writes its value in a concurrent write. We say that a language L has *parallel running time* $T(n)$ if there is a PRAM deciding L with running time $T(n)$. See the survey by Karp and Ramachandran [70] for more details.

For the PRAM model, the *unit-cost PRAM* is a PRAM where each PRAM operation is counted as a single step. The *log-cost PRAM* is a PRAM where each PRAM operation is counted as $O(\log r)$ steps, where r is the maximum size of the operands in the operation. We define $P_{unit}(f)$ and $P_{log}(f)$ to be the minimum computation time to compute f in the unit-cost and log-cost PRAM models, respectively. For our purposes, it is sufficient to consider CRCW PRAMs. Define $PRAM_{unit}(t(n)) = \{L \subseteq \{0,1\}^* : P_{unit}(L_n) = O(t(n))\}$ and $PRAM_{log}(t(n)) = \{L \subseteq \{0,1\}^* : P_{log}(L_n) = O(t(n))\}$. Similarly to circuit classes, we will also consider uniform versions of PRAM classes, where the PRAM program can be generated by a space-efficient algorithm.

There is a close connection between circuit depth and PRAM time. Stockmeyer and Vishkin [130] showed that given a CRCW Priority PRAM with $P(n)$ processors running in $T(n)$ time, it can be simulated by a family of *unbounded fan-in* circuits of size polynomial in $P(n)$ and depth $O(T(n))$. Conversely, given an unbounded fan-in circuit of size s and depth d, it can be simulated by a CRCW Priority PRAM with $O(s)$ processors running in $O(d)$ time. In the context of bounded fan-in circuits, their result implies that $PRAM_{log}(t(n)) \subseteq DEPTH(t(n)\log m(n))$ and $DEPTH(s(n)) \subseteq PRAM_{log}(s(n))$, where $m(n)$ is the maximum of $t(n)$, the number of processors, and the input length n. All simulations can be made logspace uniform. These results hold even for the unit-cost PRAM model as long as multiplication is not counted as a unit-cost instruction.

Dymond and Tompa [34] showed that $DTIME(t(n)) \subseteq PRAM_{unit}(\sqrt{t(n)})$ for the unit-cost CREW PRAM model. This also holds for logspace uniform unit-cost CREW PRAM. However, no such result is known for the log-cost PRAM model. Since $DEPTH(s(n)) \subseteq PRAM_{log}(s(n))$, improving Theorem 1.2.2 by at least a polylog factor in the logspace uniform setting would also imply non-trivial relationship between $DTIME$ and log-cost $PRAM$ computation time.

1.3.4 Circuit Size versus Formula Size

Boolean formulas can be seen as a computational model where the re-use of partial computation results is prohibited, since every gate can have fanout at most 1. However, currently we do not know if any function requires formula size that is more than the cube of its optimal circuit size.

Let $L(f)$ denote the minimum formula size for a Boolean function f. It is known that $D(f) = \Theta(\log L(f))$. By Spira's theorem (Theorem 1.2.3) $D(f) = O(\log L(f))$. The other direction is easy to see by the following argument: any circuit of depth d can be expanded to a tree-like circuit with at most 2^d leaves. This implies that $D(f) = \Omega(\log L(f))$. This shows a *tight* connection between circuit depth and the logarithm of formula size. Therefore the study of circuit size versus circuit depth is equivalent to the study of circuit size versus formula size.

1.3.5 Circuit Value Problem

The Circuit Value problem for general circuits is known to be P-complete under logspace reductions [77]. As far as we know, there is no non-trivial result about the space complexity of the Circuit Value problem for general circuits. The question of finding a uniform version of Spira's theorem has direct relevance for the complexity of the Boolean Formula Value problem. While a logspace uniform version of Spira's restructuring algorithm in Theorem 1.2.3 is still not known, it was proved (by a different approach), that for Boolean formulas presented as parenthesized expressions the Boolean Formula Value problem is in $SPACE(\log n)$ [87], and in $DLOGTIME$-uniform NC^1 [19, 17]. Improving Theorem 1.2.2 for other classes of circuits in the uniform setting will immediately improve the space complexity for the Circuit Value problem for those classes of circuits. We obtain a couple of such results after generalizing Spira's theorem to special classes of circuits including planar circuits, where the corresponding Planar Circuit Value problem is P-complete [53].

1.4 Overview of Main Results

1.4.1 Layered Circuits, Synchronous Circuits, and Circuits with Small Separators

We solve the size versus depth problem for special classes of Boolean circuits. As far as we know, previously no better bounds were known for these classes than what follows from the bounds for general circuits from Theorem 1.2.2 [101, 34]. We obtain significant improvements over these general bounds for layered circuits, synchronous circuits, and planar circuits as well as classes of circuits with small separators. Informally, a circuit is layered if its set of gates can be partitioned into subsets called layers, such that every wire between two gates in the circuit is between adjacent layers. A circuit is synchronous if for any gate g, every path from the inputs to g has the same length. The separator of a graph is a subset of the nodes whose removal yields two subgraphs of comparable sizes, where a divide-and-conquer strategy can usually be used. See Definition 3.1.3, Definition 3.1.5 , and Definition 2.4.1 in later chapters for formal definitions.

Our results are as follows. We show that every layered Boolean circuit of size s can be simulated by a layered Boolean circuit of depth $O(\sqrt{s \log s})$. Furthermore, every synchronous Boolean circuit of size s can be simulated by a synchronous Boolean circuit of depth $O(\sqrt{s})$. Lastly, every Boolean circuit of size s and separators of size $f(s)$ can be simulated by a planar Boolean circuit of depth $O(f(s))$ if $f(s) = \Omega(s^\epsilon)$ for some constant $\epsilon > 0$, or $O(f(s) \log s)$ otherwise. The last result implies that every planar Boolean circuit of size s can be simulated by a planar Boolean circuit of depth $O(\sqrt{s})$ computing the same function, with the help of the planar separator theorem [84]. Note that any Boolean circuit of size s can be converted to either a planar or a synchronous circuit of size $O(s^2)$ [146]. Thus improving these results by polylog factors would also yield improvements over the best known bounds for general circuits.

The main technique used is the two-person pebble game introduced by Dymond and Tompa [34]. For circuits with small separators such as the planar circuits, we give a divide-and-conquer pebbling strategy. On the other hand, not all synchronous circuits and layered circuits have small separators. See [117] for many examples. Furthermore, Hromkovič [60] and Gubáš, Hromkovič, and Waczulík [55] showed that there exists a sequence of Boolean functions f_n such that, f_n can be

computed by a circuit of size $O(n)$ and no size-optimal circuit for f_n has sublinear separators. This implies a limitation of the divide-and-conquer approach, since not all size-optimal Boolean circuits have sublinear separators. For synchronous circuits, our technique is to find a relatively small level such that the function can be computed by the composition of two circuits of small depths through the small level. This gives a simple proof for synchronous circuits, but the same method cannot be applied to the more general layered circuits. For layered circuits we develop an adaptive strategy in the two-person pebble game. Note that both [101] and [34] implicitly use the notion of separators in their proofs. Our results for synchronous circuits and layered circuits show that the minimum circuit depth does not necessarily grow with the separator size of the minimum-size circuit.

1.4.2 A Generalization of Spira's Theorem for Circuits with Small Separators or Segregators

The results obtained in Section 1.4.1 use the two-person pebble game, which is not uniform. See Definition 1.1.3 for the definition of uniform circuits. Informally, uniform circuit families mean that the circuit descriptions can be generated in small space. This property has further implications for the Circuit Value Problem over these circuit families. See Section 1.4.3 later.

Our main result in this direction is a uniform generalization of Spira's theorem for circuits with small separators or segregators. See Definition 2.4.1 and Definition 2.4.2 for their formal definitions. In particular, we show that if the circuit of size s has a segregator of size $f(s)$, we obtain a simulating circuit of depth at most $O(f(s)\log s)$, whose circuit description can be generated in space $O(f(s)\log^2 s)$. Furthermore, the value $f(s)$ does not have to be provided in advance. A special case of our result is that for circuits with constant-size segregators or separators, the simulating circuits of depth $(\log s)$ we obtain can be generated in space $O(\log^2 s)$.

We also observe that a variant of our technique can be applied to monotone circuits. In particular, any monotone circuit of size s that has separators or segregators of size $f()$ can be simulated by a *monotone* circuit of depth $O(f(s)\log s)$, and the description of the simulating circuit can be generated in space $O(f(s)\log^2 s)$. This generalizes Wegener's result [145], which showed that Spira's theorem [129] (Theorem 1.2.3) is still true for monotone formulas.

1.4.3 Circuit Value Problem

Our results in Section 1.4.2 generalize Spira's theorem in a uniform way. This allows us to bound the space complexity of the Circuit Value Problem for circuits with small separators and segregators. We show that the Boolean Circuit Value Problem for circuits with constant-size segregators (or separators) is in deterministic $SPACE(\log^2 n)$. Our results also imply that the Planar Circuit Value problem, which is known to be P-Complete [53], is in $SPACE(\sqrt{n}\log n)$.

In addition we show that the Layered Circuit Value and the Synchronous Circuit Value problems, which are both P-complete [54], are in $SPACE(\sqrt{n})$. However, since layered circuits and synchronous circuits do not necessarily have small separators or segregators, instead of using our generalization of Spira's theorem we use a different approach.

1.4.4 The Degree of Boolean Circuits

We consider the notion of *degree* of Boolean circuits given by Definition 6.1.1. The degree of arithmetic circuits was well studied. See Section 2.5 for a brief review. The degree of Boolean circuits was first introduced by Skyum and Valiant [128] as a measure of parallelism. We establish some basic facts about the degree of Boolean circuits regarding simulation in small circuit depth. It is easy to see that Theorem 2.5.3 by Valiant, Skyum, Berkowitz and Rackoff [137] holds for Boolean circuits. We consider two special circuit families, skew circuits and multilinear circuits, where their degrees can be bounded. We observe some implications of Theorem 2.5.3 to these classes.

1.4.5 Space-Efficient Algorithms for SUBSET SUM

One of our results in Section 1.4.2 gives a space-efficient algorithm computing the segregators of DAGs. Toward this direction, we also show that the separators of undirected graphs can be found in small space. An extension of our approach is a space-efficient algorithm for the SUBSET SUM problem. Our results are described as follows.

SUBSET SUM is the following problem: given $t \in Z^+$ and a set S of m positive integers, output YES iff there is a subset $S' \subseteq S$ such that the sum of all numbers in S' equals t. The problem was one of Karp's first 21 NP-complete

problems [69], and is known to be solvable in pseudo-polynomial time. Here we consider the search and optimization versions of SUBSET SUM. The search version asks to output S' if exists, and the optimization version asks to output a subset of S with the largest sum smaller than t over all subsets, where the largest sum is not known in advance. We give FPTAS (Fully Polynomial-Time Approximation Scheme) algorithms solving the search and optimization versions. Furthermore, the algorithms run in space $O(\frac{1}{\epsilon}\log t + \log n)$, where n is the input length and ϵ is the approximation factor. This implies a logspace approximation algorithm when the target sum t is polynomial in n and ϵ is a constant. In the special case where $\gamma = \frac{1}{3}$, we get a deterministic 2-pass streaming algorithm running in space $O(\log t + \log n)$. We then apply our techniques to the problem of finding γ-balanced separators. The γ-balanced separator of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that after removing S, both components have sizes at least $(1-\gamma)\frac{|V|}{2}$. We show that if the graph has a γ-balanced separator of size h, then it can be computed in space $O((h + \frac{1}{\gamma})\log |V|)$.

1.4.6 Simultaneous Time and Space of Input-Oblivious Turing Machines

We give two circuit simulations of input-oblivious Turing machines. First we show that languages decided by input-oblivious deterministic Turing machines in simultaneous time $t(n)$ and space $s(n)$ can be decided by families of planar circuits of size $O(t(n)s(n))$. Using this, we show that languages decided by input-oblivious deterministic Turing machines in simultaneous time $t(n)$ and space $s(n)$ can be decided in circuit depth $O(\sqrt{t(n)s(n)})$. We obtain these results by combining the constructions of oblivious Turing machines by Pippenger and Fischer [103] and Schnorr [124], the table method by Cook [24], the Planar Separator Theorem by Lipton and Tarjan [84], and our generalization of Spira's theorem (Theorem 4.1.1 in Chapter 4).

Based on our construction of planar circuits of size $O(t(n)s(n))$, we obtain quadratic lower bounds on the product of time and space for several explicit functions on input-oblivious Turing machines. In particular, we show that matrix multiplication of two $m \times m$ Boolean matrices requires time and space product $\Omega(n^2)$ on input-oblivious Turing machines, where $n = 2m^2$ is the input length. Our bound for matrix multiplication is tight. See Section 8.3.4 for details.

1.4.7 Discussion

As mentioned above, for the problem of simulating circuit size by depth, we obtain significant improvements over the general bounds for layered circuits, synchronous circuits, and planar circuits as well as classes of circuits with small separators. However, for general circuits, it is still unknown if the result by Paterson and Valiant [101] can be improved. Showing that the current $O(s/\log s)$ upper bound is tight would imply the following statement.

There exists a Boolean function $f : \{0,1\}^n$ such that f can be computed by a Boolean circuit of size s, but f cannot be computed by any Boolean circuit of depth $\Omega(s^{1-\epsilon})$ for any constant $\epsilon > 0$.

McColl and Paterson [95] showed that every Boolean function depending on n variables has circuit depth at most $n + 1$. Since ϵ is a constant, this implies that s is polynomial in n. Then f can be computed by a polynomial-size circuit, but every circuit computing f has depth $\Omega(s^{1-\epsilon})$. Then we would have a separation of NC from P in the non-uniform setting. This indicates that in case the current $O(s/\log s)$ upper bound is tight, proving this would be difficult.

Chapter 2

Previous Results

As mentioned in Chapter 1, Theorem 1.2.2 gives the best result so far for general Boolean circuits, and the simulation in Spira's theorem for Boolean formulas is tight in depth (within a constant factor). Here we give a brief sketch of each construction and discuss the techniques used in previous related results regarding the size-versus-depth question.

2.1 Pebble Games

2.1.1 The One-Person Pebble Game

Hopcroft, Paul, and Valiant [59] formally defined the one-person pebble game. The game is played on a DAG G, and it has three rules:

1. A pebble may be placed on any node with no immediate predecessors at any time.
2. A pebble may be removed from a node at any time.
3. If all the immediate predecessors of a node have pebbles on them, then a pebble may be placed on that node.

The purpose of the game is pebble the output of G, and the game stops as soon as the output is pebbled. Interested readers may see [104] and [105] for early formulations and surveys of the one-person pebble game.

For the purpose of this chapter, we will focus on playing the one-person pebble game on any DAG with bounded in-degree, which includes the Boolean circuits as previously defined.

We can think of the one-person pebble game as a tool to measure how much space is needed to carry out a computation. Intuitively, the second rule says that we can free the space once we have completed the computation, and the third rule says that a computation can be carried out only if we have computed the results it depends on. See Section 8.2 for the relations among pebble games, circuit width and breadth, and space complexity.

One classical use of the one-person pebble game is to study the time-space tradeoff of a computation (see [25], [102], and [59]). In particular, Hopcroft, Paul, and Valiant [59] established that $DTIME(t) \subseteq SPACE(t/\log t)$ (Theorem 1.3.1). Informally, their approach is to first divide the computation steps in the Turing machine into subsets according to the space where the computations occur, and build a dependency graph on the subsets. Then they give an efficient strategy in the one-person pebble game on the dependency graph, and design the space-efficient algorithm according to the pebbling strategy.

Paul, Tarjan, and Celoni [102] used superconcentrators to show that the above result cannot be improved using the one-person pebble game technique.

Theorem 2.1.1. *[102] For all $s \geq 2$, there is a graph G of size s such that pebbling some node in G requires $cs/\log s$ pebbles, where c is a constant.*

They also gave an optimal pebbling strategy that meets this lower bound.

Theorem 2.1.2. *[102] For any graph G of size s, any node of G can be one-person pebbled using $O(s/\log s)$ pebbles.*

2.1.2 The Two-Person Pebble Game

The two-person pebble game was defined in [34]. As in the one-person pebble game, the two-person pebble game is played on a DAG G. There are two players, the challenger and the pebbler. The challenger starts the game by challenging any single node of G, then the pebbler puts some pebbles on a subset of the nodes. From this point on, the challenger can only challenge a node that was either challenged or pebbled in the previous round. The game continues until at the beginning of the

pebbler's move, all the predecessors of the currently challenged node w are already pebbled. Then we say that the challenger *loses G at w*. If, under the best defense of the challenger, the pebbler can win with t number of pebble placements, then we say that G can be *two-person pebbled in time t*. The *number of rounds* in a game is equal to the number of moves by the challenger. Unlike the one-person pebble game, the pebbler does not remove pebbles once a node is pebbled. By convention, if the pebble game is played on a circuit with one output, then the challenger starts the game by challenging the output node.

In their paper, Dymond and Tompa showed that if G is a DAG of size s, then the pebbler can win the game in time $O(s/\log s)$ (Theorem 2.2.1). It was also shown that given a circuit C computing a function f, if C can be two-person pebbled with t pebbles, then there exists a tree-like circuit of depth $2t+1$ that also computes f (Theorem 2.2.2).

Here is the partial converse.

Observation 2.1.3. *Let C be a circuit computing a function f in depth d. Then the pebbler can win the two-person pebble game with at most $2d$ pebbles.*

Proof. The pebbler simply puts pebbles on the two immediate predecessors of the currently challenged node. Then the number of pebbles is at most twice the depth. □

Notice that Theorem 2.2.2 and Lemma 2.1.3 together do not give a characterization of the minimum circuit depth of a function by the two-person pebble measure, since the minimum circuit depth refers to all possible circuits for that particular Boolean function, while the two-person pebble game only considers one fixed circuit.

Using Theorem 2.2.1 and Theorem 2.2.2, Dymond and Tompa [34] gave an alternative proof of Theorem 1.2.2 originally by Paterson and Valiant [101], which is the best result for the simulation of size by depth for general circuits.

For the purpose of this chapter, we shall focus on playing the two-person pebble game only on Boolean circuits.

The following theorem was implicit in Kalyanasundaram and Schnitger [65]. They used the two-person pebble game to extend the above result to unbounded and semi-unbounded fanin (unbounded fanin for $\vee$ and fanin 2 for $\wedge$) circuits.

Theorem 2.1.4. *(Implicit in [65]) Suppose that the pebbler can win the game on a circuit C in r rounds. Then we have the following simulations.*

1. *C can be simulated by an unbounded fanin tree $T(C)$ of depth $O(r)$ and fanin $2^{O(m)}$, where m is the maximum number of pebbles over all rounds.*

2. *C can be simulated by a semi-unbounded fanin tree $T(C)$ of depth $O(r \log m)$ and fanin $2^{O(m)}$, where m is the maximum number of pebbles over all rounds.*

The following theorem by Tompa [132] relates the one-person and the two-person pebble games.

Theorem 2.1.5. *[132] If a graph G can be two-person pebbled in time T then G can be one-person pebbled with $T + 1$ pebbles.*

2.2 General Boolean Circuits

2.2.1 The Construction by Paterson and Valiant

Paterson and Valiant [101] gave the first proof of Theorem 1.2.2. Their idea is to partition the circuit C into two subcircuits C_1 and C_2, such that the sizes of C_1 and C_2 differ by at most 1, C_1 contains the output of C, and there are no paths from C_1 to C_2. It is easy to see that such partition exists. If this partition cuts many gates in C, then recurse on C_1 and C_2. Otherwise suppose that the partition cuts C at gates $g_1, \ldots, g_m$. We can compute the disjunctive normal form of the function computed at the output of C in terms of the functions computed at $g_1, \ldots, g_m$. Then we compute the disjunctive normal form and the subcircuits computing $g_1, \ldots, g_m$ in parallel. Since m is small (e.g. $m = \Theta(s/\log s)$), we will get a circuit of smaller depth in this case.

2.2.2 The Construction by Dymond and Tompa

Dymond and Tompa [34] proved Theorem 1.2.2 using the two person pebble game. The two-person pebble game is also used to simulate the Boolean circuit by an alternating Turing machine. See [34] for details. Several other variants of pebble games have been invented to study questions related to the space requirements of computation, e.g. [107, 25]. See [104] for a survey, and [43, 14, 13, 27] for recent

advancements. Here we will focus on the two-person pebble game. See Section 2.1.2 for the definition of the two-person pebble game.

The next two theorems by Dymond and Tompa [34] give an alternative proof of Theorem 1.2.2.

Theorem 2.2.1. *[34] Let G be a DAG with node set V. Then the pebbler can win the game in time $O(|V|/\log|V|)$.*

Theorem 2.2.2. *[34] Let C be a Boolean circuit computing a function f. If the underlying graph of C can be two-person pebbled with t pebbles, then there exists a tree-like circuit of depth $2t+1$ that also computes f.*

2.3 Boolean Formula: The Construction by Spira

Jordan [64] first gave the combinatorial lemma stating that a tree T has a node v such that after removing v from T, T is divided into two subtrees T_1 and T_2, such that the sizes of both are a constant fraction of the size of T, e.g. $|T_1| \leq \frac{2}{3}|T|$ and $|T_2| \leq \frac{2}{3}|T|$. Spira [129] proved Theorem 1.2.3 using this fact. Let F be the original Boolean formula, and let v be the node in Jordan's lemma. Let F_v be the subformula of F under v, and let F_0 and F_1 be the two formulas obtained from F by replacing v by 0 and 1, respectively. It is easy to see that for any input x, we have

$$F(x) = (F_1(x) \wedge F_v(x)) \vee (F_0(x) \wedge \neg F_v(x)). \tag{2.1}$$

The construction recurses on F_v, F_0, and F_1. Since the sizes of the formulas decrease by a constant factor in each recursive step, at the end of the recursion, we will get a Boolean formula of logarithmic depth and polynomial size (in terms of the size of F).

As stated in Section 1.2, there are several follow-up results of Theorem 1.2.3, including Bonet and Buss [10] and Wegener [145]. All these results are based on Spira's construction.

2.4 Separators and Segregators

Informally, a node separator of a graph G is a set of nodes whose removal yields two disjoint subgraphs of G. In this section we only consider *balanced* separators, that

yield subgraphs that are comparable in size. In the next definition each of the two subDAGs could consist of several weakly connected components.

Definition 2.4.1. A *separator of size k* of a DAG $G = (V, E)$ is a set of k nodes $S \subseteq V$ such that $G \setminus S$ is not weakly connected (i.e. the underlying undirected graph is not connected); and the removal of S partitions $G \setminus S$ into two subDAGs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, such that $|V_i| \leq \frac{2}{3}|V|$ for $i = 1, 2$, and there are no edges either from G_1 to G_2, or from G_2 to G_1 in $G \setminus S$.

Note that the two subDAGs could be disconnected within themselves.

Segregators are a relaxation of separators in directed acyclic graphs [144, 119].

Definition 2.4.2. [144, 119] A *segregator of size k* of a DAG $G = (V, E)$ is a set of k nodes $S \subseteq V$ such that every node in $G \setminus S$ has at most $\frac{2}{3}|V|$ predecessors in $G \setminus S$.

Definition 2.4.3. We say that a Boolean circuit C has separators of size $f()$ if the underlying DAG of every subcircuit of C with s gates has a separator of size at most $f(s)$. We say that a Boolean circuit C has segregators of size $f()$ if the underlying DAG of every subcircuit of C with s gates has a segregator of size at most $f(s)$.

The above definition is reasonable, since we typically consider classes of circuits based on properties of their underlying DAGs that are closed with respect to subDAGs, for example planar circuits, circuits with small treewidth, etc.

We talk about constant-size separators (resp. segregators), if the size of the separator (resp. segregator) is bounded by a fixed constant that does not depend on the size of the circuit.

The following lemma follows directly from the definitions.

Lemma 2.4.4. *Any DAG with a separator of size k has a segregator of size k.*

Proof. Let S be a separator of G of size k, and S partitions $G \setminus S$ into G_1 and G_2. Let v be any node in $G \setminus S$. Let $P(v)$ be the set of predecessors of v in $G \setminus S$. Since S is a separator of G, we must have either $P(v) \subseteq G_1$ or $P(v) \subseteq G_2$. Then S is also a segregator of G of size k. □

Notice that the reverse is not true in general, since a node in a DAG may have a smaller number of predecessors than the size of the component that contains the node in the underlying undirected graph.

Upper bounds on the separator size of several natural graphs are known. Jordan [64] first proved that trees have separators of size 1.

Theorem 2.4.5. *[64] Let T be a tree. Then T has a separator of size* 1.

Lipton and Tarjan [84] proved the following theorem for the planar graphs.

Theorem 2.4.6. *[84] Given a planar graph G of size s, then G has a separator of size $O(\sqrt{s})$.*

Gilbert, Hutchinson, and Tarjan [51] gave the following theorem for graphs of bounded genus. The *genus* of a graph is the least number of handles we add to the plane so that the graph can be embedded on the plane with handles without crossing edges.

Theorem 2.4.7. *[51] Given a graph G of size s and genus g, then G has a separator of size $O(\sqrt{gs})$.*

A *minor* of a graph is obtained by identifying two nodes connected by an edge, and then removing any self-loops. Let K_k denote the complete graph on k nodes. For graphs having no K_k as a minor, we have the following theorem due to Alon, Seymour, and Thomas [5].

Theorem 2.4.8. *[5] Given a graph G with node set V and having no K_k as a minor, G has a separator of size $O(k^{\frac{3}{2}}|V|^{\frac{1}{2}})$.*

2.5 Arithmetic Circuits

In this section we define the arithmetic circuit model and review the previous results about size versus depth on the arithmetic circuits. An arithmetic circuit is similar to a Boolean circuit defined in Section 1.1.2 but with inputs from a semi-ring, and $\wedge$ and $\vee$ replaced by $\times$ and $+$, respectively. Formally, an arithmetic circuit over a commutative semi-ring $R = (R, +, \times, 0, 1)$ is a labeled directed acyclic graph (DAG), where every node is labeled by an element in R, a variable from $\{x_1, \ldots, x_n\}$, or a semi-ring operation $+$ or $\times$. As in the Boolean circuits, the *inputs* and *output* of

an arithmetic circuit are the nodes with in-degree 0 and out-degree 0, respectively. The nodes with non-zero in-degree are called *gates*. A formula (or tree-like circuit) is a circuit whose fanout is one for every gate except the output. The size of an arithmetic circuit is the number of its gates. The *depth of a gate* g is the length of the longest path from any input to g. The *depth of an arithmetic circuit* C is the depth of the output gate.

The two surveys by Allender [3], and Shipilka and Yehudayoff [127] described the main difference between arithmetic circuits and Boolean circuits. The Boolean circuits can perform operations on the individual bits of the inputs such that the operations are not arithmetic. On the other hand, arithmetic circuits cannot since the gates are labeled by the arithmetic operations $\times$ and $+$ only. Also, since arithmetic operations can be simulated efficiently by Boolean circuits, a lower bound on the size of Boolean circuits implies a lower bound on the size of arithmetic circuits, while the other direction does not necessarily hold. Furthermore, Allender [3] mentioned that in the version of the arithmetic circuit model where the circuits have access to each individual bit, there are cases where this model may be either more powerful or less powerful than Boolean circuits.

The *degree* of arithmetic circuits is defined as follows. Notice that in the literature, the terms "degree" and "algebraic degree" are used interchangeably. We will adopt the term *degree* unless stated otherwise.

Definition 2.5.1. The *degree of a node* in an arithmetic circuit is defined inductively as follows.

- The degree of a node labeled by an element in R is 0.
- The degree of a node labeled by a variable is 1.
- The degree of a $+$ node is the maximum degrees of its children.
- The degree of a $\times$ node is the sum of the degrees of its children.

The *degree of an arithmetic circuit* is the maximum over the degrees of all its nodes.

For simulating arithmetic circuits with smaller depth, Brent [15] gave the first result for arithmetic formulas.

Theorem 2.5.2. *[15] Let C be an arithmetic circuit over the field of real numbers with additions and multiplications. Let s be the size of C. Then C can be simulated by an arithmetic circuit C' over the same field such that the depth of C' is $O(\log s)$.*

Valiant, Skyum, Berkowitz and Rackoff [137] generalized Brent's result to arbitrary arithmetic circuits over a semiring.

Theorem 2.5.3. *[137] Let C be an arithmetic circuit over a commutative semiring such that C has size s and degree d. Then there exists an arithmetic circuit C' computing p such that C' has size $O(s^3)$ and depth $O(\log s \log d)$.*

Miller, Ramachandran, and Kaltofen [98] later gave the following theorem.

Theorem 2.5.4. *[98] Let C be an arithmetic circuit over a commutative semi-ring such that C has size s and degree d. Then on any input, the value of C can be evaluated by a parallel algorithm in parallel time $O(\log s(\log s + \log d))$ using $M(s)$ processors, where $M(s)$ is the number of processors to multiply two $s \times s$ matrices over the same semi-ring in $O(\log s)$ parallel time.*

Note that the construction in [137] is not uniform, while [98] gave an explicit PRAM algorithm. Furthermore, Theorem 2.5.3 needs to know the degree of C in advance, while Theorem 2.5.4 does not.

Allender, Jiao, Mahajan, and Vinay [4] proved the following theorem for log-space uniform circuits.

Theorem 2.5.5. *[4] Let R be any commutative semiring. The class of functions computed by* log*space uniform arithmetic circuits over R of size s and degree d is equal to the class of functions computed by $O(\log s + \log d)$-space uniform arithmetic circuits over R of size polynomial in s and d and depth $O(\log s \log d)$.*

In fact the proofs of Theorem 2.5.3 and Theorem 2.5.5 give an even stronger result. Here we state the uniform version. Recall that a Boolean (resp. arithmetic) circuit is semi-unbounded fanin if the fanin of $\vee$ (resp. $+$) gates is unbounded, while the fanin of $\wedge$ (resp. $\times$) is bounded. In particular, SAC^1 is the class of functions computed by semi-unbounded fanin Boolean circuit of size $n^{O(1)}$ and depth $O(\log n)$.

Theorem 2.5.6. *[4] Let R be any commutative semiring. The class of functions computed by* log*space uniform arithmetic circuits over R of polynomial size and polynomial degree is equal to the class of functions computed by* log*space uniform semi-unbounded fanin arithmetic circuits over R of polynomial size and depth $O(\log n)$.*

We write VP to denote the class of arithmetic circuits of polynomial size and polynomial degree, and VNC^i to denote the class of arithmetic circuits of polynomial size, polynomial degree, and $O(\log^i n)$ depth. Then Theorem 2.5.3 and Theorem 2.5.4 imply that $VP = VNC^2$.

Nisan and Wigderson [99] first defined *multilinear* arithmetic circuits. A polynomial f is multilinear if the degree of each variable in f is at most one. An arithmetic circuit C is multilinear if every gate in C computes a multilinear polynomial. Later Raz [111] defined *syntactically multilinear* arithmetic circuits. An arithmetic circuit C is syntactically multilinear, if for every $\times$ gate $v \in C$ with immediate children v_1 and v_2, the two sets of circuit inputs of the subcircuits of v_1 and v_2 are disjoint. Raz and Yehudayoff [113] showed that the construction by Valiant, Skyum, Berkowitz and Rackoff [137] for Theorem 2.5.3 preserves syntactic multilinearity. Here is the formal statement.

Theorem 2.5.7. *[113] Let C be a syntactically multilinear arithmetic circuit of size s and degree r over the field F and over the set of variables $X = \{x_1, \ldots, x_n\}$ computing the polynomial f. Then there exists a syntactically multilinear arithmetic circuit C of size $O(s^3 d^6)$, depth $O(\log s \log d)$, and degree d over the field F and over the set of variables X computing f. In particular, syntactically multilinear VP equals syntactically multilinear VNC^2.*

2.6 Uniformity

Let s be the size of the circuit in Theorem 1.2.2. In the worst case, the construction by Paterson and Valiant gives a circuit of size $2^{\Theta(\frac{s}{\log s})}$ and depth $\Theta(\frac{s}{\log s})$. A direct implementation uses linear space to find the partition in each recursion. Since the size of the resulting circuit is $2^{\Theta(\frac{s}{\log s})}$, we also need a counter of size $\Theta(\frac{s}{\log s})$ to keep track of the position.

Consider the uniformity of the construction by Dymond and Tompa. Since

the two-person pebble game is played on the circuit, we need a Turing machine to simulate the two-person pebble game to generate the simulating circuit. Very recently Chan [21] showed that the two-person pebble game is PSPACE-complete. Some variants of the pebble game are also known to be PSPACE-complete, e.g. the one-person black pebble game [52] and the one-person black and white pebble game [58].

It is not known if the restructuring algorithm in Spira's construction can be implemented in logspace [17]. Thus it is not known if the simulating circuit families in either of these constructions can be made uniform in small space.

2.7 The Karchmer-Wigderson Game and Circuit Depth

We briefly review the Karchmer-Wigderson communication game [68]. Let f be a Boolean function. There are two players in the game, the 0-player and the 1-player. The two players can communicate with each other by sending bits. The 0-player and the 1-player are given x and y, such that $f(x) = 0$ and $f(y) = 1$, respectively. The goal of the game is then to define a communication protocol such that at the end of the game, the two players agree on an integer i such that $x_i \neq y_i$. See Kushilevitz and Nisan [76] for more details. The significance of the Karchmer-Wigderson communication game is that the minimum number of bits sent by the two players in the game *equals* the minimum circuit depth of f. In the following we give the formal statements.

Definition 2.7.1. For a Boolean function $f : \{0,1\}^n \to \{0,1\}$ let $X = f^{-1}(x)$ (i.e. the set of all x such that $f(x) = 1$) and $Y = f^{-1}(0)$. Let $R_f \subseteq X \times Y \times \{1, \ldots, n\}$ consist of all triples (x, y, i) such that $x_i \neq y_i$.

Theorem 2.7.2. *[68] Let $d(f)$ denote the minimum depth of all Boolean circuits computing f, and let $D(R_f)$ denote the minimum number of bits of all protocols for R_f defined above. Then $d(f) = D(R_f)$.*

Thus we can approach the problem of simulating circuit size by smaller circuit depth as follows. Given a Boolean circuit C of size s, design a communication protocol in the Karchmer-Wigderson game such that the number of bits exchanged by the two players in the protocol is as small as possible. That is, the 0-player and the 1-player receive x and y such that C evaluates to 0 and 1 on x and y,

respectively, and the two players need to find an index i such that $x_i \neq y_i$. However, it is unknown if the construction of the simulating circuits in Theorem 2.7.2 can be made uniform.

2.8 Circuit Value Problem

The Boolean Circuit Value problem (CVP) is defined as follows: given the standard description of a circuit C and an assignment x to the variables of C as the input, compute the value of the output of the circuit C evaluated on the assignment x. For P-complete problems, Ladner [77] showed that CVP is P-complete. Goldschlager [53] showed that the Planar Circuit Value problem (PCVP) is also P-Complete. Greenlaw, Hoover, and Ruzzo [54] showed that the Synchronous Alternating Monotone Fanin 2 Fanout 2 (SAM2CVP) is P-complete, which implies that the less restricted Synchronous Circuit Value and the Layered Circuit Value problems are both P-complete. Vitter and Simons [142] showed that CVP over unbounded fanin circuits with more than a linear order of wires is also P-complete.

For CVP over arguably more restricted classes, the Boolean Formula Value Problem (the Circuit Value problem for tree-like circuits) can be solved in DLOGTIME-uniform NC^1 when the formula is presented in parenthesized expression [19, 17, 87]. However, when the Boolean formulas are presented as tree-like circuits, the best result so far shows only that the Boolean Formula Value Problem can be solved in $O(\log^2 n)$ space. For the Circuit Value problem over the Comparator Circuit class, Mayr and Subramanian [94] gave an $O(\sqrt{n}\text{poly}\log(n))$ time PRAM algorithm.

In the following we review results about Monotone Planar Circuit Value Problem (MPCVP) on space-bounded Turing machines and PRAMs. It is known that $NC^k \subseteq EREW^k \subseteq CREW^k \subseteq CRCW^k = AC^k \subseteq NC^{k+1}$. See the survey by Karp and Ramachandran [70]. Goldschlager [53] first showed that the Upward Stratified MPCVP can be solved in space $O(\log^2 n)$, where upward stratified circuits are layered circuits such that every wire can be embedded towards the output, and all circuit inputs belong to the same layer. Dymond and Cook [33] showed that the problem is in $LOGCFL$. Later Barrington, Lu, Miltersen and Skyum [6] improved to $LOGDCFL$. A less restricted problem is the Layered Upward MPCVP, where circuit inputs could be in different layers. Kosaraju [73] gave an $O(\log^3 n)$ time CREW algorithm for this problem using polynomial number ($\Omega(n^6)$) of processors.

Ramachandran and Yang [110] gave an $O(\log^2 n)$ time EREW algorithm using only linear number of processors. Another variant of the Upward Stratified MPCVP is the One-Input-Face MPCVP, where all circuit inputs belong to the same face on the plane. Yang [151] gave an $O(\log^2 n)$ time EREW algorithm using polynomial number of processors. Recently Limaye, Mahajan, and Sarma [79] showed that the problem is in $LOGCFL$. Chakraborty and Datta [20] further showed that the problem is in $LOGDCFL$ and is hard for logspace. For the most general MPCVP, Delcher and Kosaraju [30] first gave an $O(\log^4 n)$ time CREW algorithm. Yang [151] showed an $O(\log^3 n)$ time EREW algorithm. Both algorithm uses polynomial number ($\Omega(n^6)$) of processors. Ramachandran and Yang [109] later gave an $O(\log^6 n)$ time EREW algorithm using exact n processors. Recently Limaye, Mahajan, and Sarma [79] showed that the problem is in $AC^1(LOGCFL) = SAC^2$, where $AC^1(LOGCFL)$ is the class of languages decided by AC^1 circuit families with $LOGCFL$ oracle gates. See Vollmer [143] for definitions. None of the Circuit Value Problems over Boolean formulas, comparator circuits, and monotone planar circuits is believed to be P-complete.

It is known that some P-complete problems can be solved in sublinear parallel time with polynomial number of processors. A circuit is *square* if the width of every layer is equal to the depth of the circuit. Condon [23] defined the notion of *strictly $T(n)$-complete* for P. Roughly, a problem L in P is strictly $T(n)$-complete for P if L has parallel running time within a polylog factor of $T(n)$ and furthermore, if this could be improved by a polynomial factor, say n^ϵ, then we can improve the parallel running time of all problems in P with at least linear running time by a $O(1/n^\epsilon)$ multiplicative factor. Condon [23] showed that the Square Circuit Value problem is P-complete, and gave a PRAM algorithm running in $O(\sqrt{n}\text{poly}\log(n))$ parallel time. Condon [23] also showed that the Lex First Maximal Clique (LFMC) problem, which is P-complete, can be solved in $O(\sqrt{n}\text{poly}\log(n))$ parallel time. Vitter and Simons [142] showed that CVP over unbounded fanin circuits with more than a linear order of wires, which is also P-complete, can be solved in $O(\sqrt{n}\text{poly}\log(n))$ parallel time. It is still unknown if the CVP over general circuits can be solved in sublinear parallel time (resp. circuit depth). Condon's results [23] as well as Vitter and Simon's results [142] yield polynomial-size and $O(\sqrt{n}\text{poly}\log(n))$-depth circuit families for these problems. Note that using Borodin's simulation of depth by space [11], the above results on PRAMs imply $O(\sqrt{n}\text{poly}\log(n))$-space algorithms. However, the

small-space simulation using Borodin's result does not yield polynomial running time.

Chapter 3

Size versus Depth via The Two-Person Pebble Game

In this chapter, we solve the size versus depth problem for special classes of Boolean circuits including layered circuits, synchronous circuits, and planar circuits as well as circuits with small separators. As far as we know, previously no better bounds were known for these classes than what follows from the bounds for general circuits from Theorem 1.2.2 [101, 34].

The main technique used is the two-person pebble game introduced by Dymond and Tompa [34]. For circuits with small separators such as the planar circuits, it is natural to design a pebbling strategy based on the divide-and-conquer method. On the other hand, not all synchronous circuits and layered circuits have small separators. As mentioned in Section 1.4.1, our idea is to consider *cuts*, which is a relaxed notion of graph separators. A cut separates the graph into two subgraphs that are not necessarily comparable in size. For synchronous circuits, our technique is to find a relatively small cut such that the function can be computed by the composition of two circuits of small depths. For layered circuits, we develop an adaptive strategy in the two-person pebble game, such that the sizes of the cuts are taken into account during the game. Note that both [101] and [34] implicitly use the notion of separators in their proofs. Our results for synchronous circuits and layered circuits show that the minimum circuit depth does not necessarily grow with the separator size of the minimum-size circuit.

Turán [134] showed that there exists a function f_n such that any synchronous

circuit for f_n has size $\Omega(n \log n)$, but there exists a layered circuit for f_n with size $O(n)$. See Belaga [8] for the same gap for functions with multiple outputs. This distinguishes synchronous circuits and layered circuits with respect to their computational powers. Notice that every Boolean function can be computed by circuits from each of the classes we consider.

Synchronous and planar circuits have been extensively studied before. Synchronous circuits were introduced by Harper [57]. Planar circuits were introduced by Lipton and Tarjan [83]. A circuit is planar if its underlying graph can be embedded in the plane without crossings of the wires. Layered circuits are a natural generalization of synchronous circuits, but as far as we know they have not been explicitly studied. Layered graphs have been studied by Paul, Tarjan, and Celoni [102] (they call these "level graphs" in their paper). Belaga [8] defined locally synchronous circuits, which is a subclass of layered circuits, with the extra condition that each input variable can appear at most once.

Most of the results in this chapter can be found in Gál and Jang [44, 45].

3.1 Layered Circuits and Synchronous Circuits

Definition 3.1.1. Height Let C be a circuit with one output, and let g be any node in C. The *height of* g is the length of the longest path from g to the output.

Recall from Definition 1.1.1 that given a circuit C with one output and a node $g \in C$, the depth of g is the length of the longest path from any input to g, and the depth of C is the depth of the output gate.

Definition 3.1.2. Levels and layers The *ith level* of a circuit consists of all gates with depth equal to i. For circuits with one output, the *ith layer* of the circuit consists of all gates with height equal to i.

Note that the levels and layers of a circuit include gates only. We do not count circuit inputs in the levels or layers.

Definition 3.1.3. [57] **Synchronous circuits** A circuit is *synchronous* if for any gate g, all paths from the inputs to g have the same length.

Note that in the definition, the circuit could have either single output or multiple outputs.

Lemma 3.1.4. *Let C be a Boolean circuit. If C is synchronous, then every wire connecting two gates in C is between adjacent levels.*

Proof. Suppose that C is synchronous. We assign levels to each node by the depth of the node. Let (g, h) be any wire in C, where g and h are two gates. Suppose that g is in the ith level. Then all paths from the inputs to g have length i, and there exists at least one path from the input to h of length $i + 1$ through the wire (g, h). Since all paths from the inputs to h have equal length, h must be in the $(i + 1)$th level, which is adjacent to the ith level. □

Definition 3.1.5. Layered circuits A circuit is *layered*, if the set of gates can be partitioned into subsets called *layers*, such that every wire connecting two gates in the circuit is between adjacent layers. For circuits with one output, the following is an equivalent definition: A circuit with one output is *layered* if for any gate g all paths from g to the output have the same length.

The following lemma shows that the two definitions for layered circuits are equivalent for circuits with one output.

Lemma 3.1.6. *Let C be a circuit with one output. Then C is layered (i.e. every wire connecting two gates is between adjacent layers) iff for any gate g, all paths from g to the output have the same length.*

Proof. Let $C = (V, E)$ be any circuit. First we show that if every path from any gate to the output has the same length, then C is layered. Define the layers $L(i)$ of V inductively as follows.

- The output of C is in $L(0)$.
- For any edge $(v, w) \in E$ such that v and w are both gates and $w \in L(i)$, we let $v \in L(i + 1)$.

We want to show that every wire connecting two gates is between adjacent layers. Since C is weakly-connected, any gate in C is in some $L(i)$. We now show that any gate belongs to exact one $L(i)$ for some i by induction. It is obvious that the output of C cannot be in $L(i)$ for $i \neq 0$. Let d be the depth of C. Consider any $(v, w) \in E$ such that v and w are gates and $w \in L(i)$, we then have $v \in L(i + 1)$. So one path from v to the output has length $i + 1$. Now if $v \in L(j)$ for some $j \neq i + 1$, then

there must exist another path from v to the output with length $j \neq i+1$, which is a contradiction. So C is layered.

Now we show that if C is layered, then every path from any fixed gate to the output has the same length. Let $L(i)$ be defined as above. Our claim is obviously true for the output of C. Now let $(v, w) \in E$ such that v, w are gates and $w \in L(i)$, then $v \in L(i+1)$. By induction hypothesis, all paths from w to the output have length i. Since any wire connecting two gates must be between adjacent layers, all edges with w as one end must have some node from $L(i+1)$ as another end. This means that all paths from v to the output have length $i+1$. □

The following lemma shows that every synchronous circuit is layered, but not vice versa. A simple counter-example would be the circuit with inputs x_1, x_2, x_3, gates $g_1 = x_1 \wedge x_2$, $g_2 = g_1 \wedge x_3$, and g_2 being the output gate.

Lemma 3.1.7. *Let C be a synchronous circuit. Then C is also a layered circuit.*

Proof. Since C is synchronous, every path from any input to the output gate has the same length, say d, and every path from any input to gate g also has the same length, say d_g. Then every path from g to the output gate has length $d - d_g$. So C is layered. □

Belaga [8] defined locally synchronous circuits, which is a subclass of layered circuits, with the extra condition that each input variable can appear at most once. Synchronous circuits form a proper subset of layered circuits by Lemma 3.1.7 and the above example. Furthermore, Turán [134] showed that there exists a function f_n such that any synchronous circuit for f_n has size $\Omega(n \log n)$, but there exists a layered circuit for f_n with size $O(n)$. See Belaga [8] for the same gap for functions with multiple outputs. This distinguishes synchronous circuits and layered circuits with respect to their computational powers.

3.2 The One-Person Pebble Game versus the Two-Person Pebble Game

Tompa [132] proved Theorem 2.1.5, which states that the number of pebbles used in the one-person pebble game is at most 1 plus the number of pebbles used in the two-person pebble game. However, we do not know much about the converse.

Conjecture 3.2.1. If a circuit C can be one-person pebbled with p pebbles, then C can be two-person pebbled in time $p \log |C|$.

On the other hand, we can show a relationship between the progressive one-person pebble game and the two-person pebble game. A *progressive* one-person pebble game is a variant of the one-person pebble game such that every node is pebbled at most once. It was used by Sethi [126] and Lengauer[78] to study the space requirement where re-computations are not allowed, i.e. most time-efficient computations.

We need the following property of the progressive one-person pebble game.

Lemma 3.2.2. *Let C be a circuit. Let t_1 be the time in the progressive one-person pebble game where exactly half of the nodes in C have not been pebbled yet. At time t_1, let T be the pebbled nodes, let U be the set of nodes that are pebbled and have their pebbles removed before t_1, and let W be the set of nodes that are not pebbled yet. If $U \neq \emptyset$, then T is a cut of C between U and W.*

Proof. Suppose that $U \neq \emptyset$. Note that T, U, and W are disjoint, and their union is C. Assume that T is not a cut between U and W. Then there must exist an edge (u, w) such that $u \in U$ and $w \in W$. In order to pebble the output of C, w must be pebbled (assuming that the output depends on every gate). This means that u must be pebbled as well because u is an immediate predecessor of w. But since the game is progressive, u cannot be pebbled twice, which means that the output cannot be pebbled. A contradiction. So T must be a cut between U and W. □

Note that in the general one-person pebble game, the above argument is not necessarily true since we may pebble the same node several times. Also, the sizes of U and W in the lemma might not be comparable to each other. Finally, notice that it is possible to generalize the lemma to any time instance other than t_1, but the lemma already suffices for our purposes.

Theorem 3.2.3. *If a circuit C can be pebbled using p pebbles in the progressive one-person game, then C can be two-person pebbled in time $O(p \log \frac{|C|}{p})$.*

Proof. Let T be the pebbled nodes in the progressive one-person pebble game at the time instance where half of the nodes are not pebbled yet. At the beginning of the two-person pebble game, the pebbler puts pebbles on T. Now if the challenger

stays on the output of C, then the pebbler puts pebbles on nodes that are pebbled in the one-person game at the time instance where $\frac{1}{4}$ of the nodes are not pebbled. If the challenger moves to some node in T, then the pebbler chooses $\frac{3}{4}$ instead of $\frac{1}{4}$. Let C_i be the subcircuit of C whose output is the challenged node in the ith round. Then the pebbler does similar for each round until the size of C_i is p, and the pebbler simply pebbles the whole C_i. So in every round before the last one, the pebbler either selects a cut for the subcircuit by Lemma 3.2.2, or the least half topologically-ordered gates of the subcircuit if no pebbles were removed in the corresponding progressive one-person pebble game. At the end of the game, the two immediate predecessors of the challenger must be pebbled, and the number of pebbles used is at most $p \log \frac{|C|}{p} + p = O(p \log \frac{|C|}{p})$. □

3.3 Simulation of Layered Circuits of Size s in Depth $O(\sqrt{s \log s})$

We begin by the following lemma similar to Observation 2.1.3.

Lemma 3.3.1. *Let C be a circuit of depth d, then there exists a tree-like circuit F that computes the same function as C, and the pebbler can win the two-person game on F with $O(d)$ pebbles.*

Proof. We first convert C to a tree-like circuit F as follows. Starting from the output of C, if the two subcircuits of its two immediate children are not disjoint, then we make copies of one of the subcircuit. Then we proceed inductively on the two subcircuits until the circuit inputs are reached. The depth of F is d, same as the depth of C. Now the pebbler simply pebbles the two predecessors of the challenged vertex in each round. Then the number of pebbles used would be at most twice the longest path in the tree-like circuit F, which is $2d$. □

The following theorem plays one-person pebble game on layered DAGs.

Theorem 3.3.2. *[102] Let G be a layered DAGs of size s, and the degree of each vertex is no more than d. Then any vertex of G can be one-person pebbled using $\sqrt{8(d-1)s}$ pebbles.*

The result is tight (depends on d) since Cook's pyramid graph [25] can be easily adapted to a layered DAG, which requires $\sqrt{s}$ pebbles.

We now present a corresponding result for the two-person pebble game. The following lemma gives an adaptive strategy in the two-person pebble game for layered circuits.

Lemma 3.3.3. *Let C be a layered circuit of size s. Then C can be two-person pebbled in time $O(\sqrt{s \log s})$. That is, the pebbler can win by using $O(\sqrt{s \log s})$ pebbles.*

Proof. First note that at any point in the game, we only need to consider the subcircuit whose single output is the currently challenged node. Thus, in the proof we can assume without loss of generality that the circuit C has only one output, and that the first move of the challenger is to challenge the output gate.

Let $L(0), \ldots, L(d)$ be all the layers, where d is the depth of C, and $L(i)$ is the set of gates with height i. (See Definition 3.1.1 for the definition of height.) Note that $L(0)$ consists of the output gate. We say that a layer $L(i)$ is large if $|L(i)| > y$ and small otherwise. We shall determine the value of y later.

The strategy of the pebbler has two phases. During the first phase, the pebbler forces the challenger to go into a subcircuit between two small layers such that every layer between the two small layers is large, or into a subcircuit such that all nodes of the subcircuit belong to large layers. In the second phase, the pebbler will win the game within that subcircuit.

During the first phase, the pebbler always pebbles a small layer S_α with $\alpha > \beta$, where S_β is the layer where the challenged node resides in that round. The pebbler continues this phase until the small layer S_α with $\alpha > \beta$ closest to the challenged node is pebbled, or until there are no more such small layers. Note that the pebbler pebbles the small layers $S_0, S_1, \ldots, S_m$ in a divide-and-conquer way depending on the location of the challenged node in each round. Since there are at most s small layers, the number of pebbles used in the first phase is at most $y\lceil \log s \rceil$.

Phase I. Let $S_0, S_1, \ldots, S_m$ be the small layers numbered starting from the output. Note that $S_0 = L(0)$ since $L(0)$ contains only one gate. Define $h(j)$ to be the height of the gates in the jth small layer S_j. We shall define the strategy inductively.

At the beginning of the game (Round 1), the challenger challenges the output node, which belongs to $S_0 = L(0)$.

Suppose that for $r \geq 1$ at the beginning of Round r the challenger challenges

a node $w \in S_j$. Note that since during Phase I pebbles are only placed on nodes in small layers, the challenged node w belongs to a small layer in every round within Phase I. We have three cases.

1. No small layer $L(h(b)) = S_b$ with $b > j$ exists. That is, every layer $L(k)$ with $k > h(j)$ is a large layer. Then the pebbler continues with the second phase.

2. None of the small layers $L(h(b)) = S_b$ with $b > j$ is pebbled. The pebbler then puts a pebble on each node of $S_{\lceil \frac{m+j}{2} \rceil}$.

3. There exists a small layer $L(h(b)) = S_b$ with $b > j$, such that all nodes in S_b are pebbled, and none of the small layers between S_j and S_b is pebbled. The pebbler then puts a pebble on each node of $S_{\lfloor \frac{b+j}{2} \rfloor}$ if $\lfloor \frac{b+j}{2} \rfloor \neq b$. If $\lfloor \frac{b+j}{2} \rfloor = b$, then there are no small layers between S_j and S_b, and the pebbler continues with the second phase.

Phase II. The pebbler's strategy in the second phase is as follows: Suppose that the challenger challenges node w in the beginning of the kth round for some k. Then the pebbler puts pebbles on the two inputs of w, say u and v. In the $(k+1)$st round, if the challenger stays on w, then the pebbler wins the game. On the other hand, if the challenger challenges one of the inputs of w, without loss of generality say u, then the pebbler puts pebbles on the two inputs of u in the $(k+1)$st round. The game continues inductively this way until at the beginning of pebbler's move either the currently challenged node w is an input of C, or the two immediate predecessors of w are already pebbled. Thus the pebbler wins in the second phase.

Note that in this phase, the pebbler only spends at most two pebbles in each round, and the two pebbles are put on nodes in large layers of C. Moreover, during k rounds of the second phase, the pebbler pebbles nodes from k different large layers. Since the number of large layers in C is at most $\frac{s}{y}$, the second phase must terminate in at most $\frac{s}{y}$ rounds. Thus, the number of pebbles used in this phase is at most $\frac{2s}{y}$.

The total number of pebbles used throughout the game is at most $p = y\lceil \log s \rceil + 2s/y$. The minimum of this expression is $p = 2\sqrt{2s\lceil \log s \rceil}$, achieved when $y = \sqrt{\frac{2s}{\lceil \log s \rceil}}$. This proves the lemma. □

Theorem 3.3.4. *Every layered Boolean circuit of size s can be simulated by a layered Boolean circuit of depth $O(\sqrt{s \log s})$ computing the same function.*

Proof. Follows immediately from Theorem 2.2.2 and Lemma 3.3.3. □

3.4 Simulation of Synchronous Circuits of Size s in Depth $O(\sqrt{s})$

The following simple lemma was given in [146]. The results by McColl and Paterson [95], and Gaskov [48] give stronger results. But for our purposes, this slightly weaker bound is sufficient, and we include a simple proof for completeness.

Lemma 3.4.1. *[146] For every function $f : \{0,1\}^n \to \{0,1\}$, there exists a synchronous circuit of depth at most $n + \log n + 1$ computing f.*

Proof. The proof is based on considering any DNF of f. The terms can be computed in parallel with depth at most $\log n + 1$, and the number of terms is at most 2^n. This gives the desired depth. Note that any circuit can be made synchronous without increasing its depth. □

Theorem 3.4.2. *Every synchronous Boolean circuit of size s can be simulated by a synchronous Boolean circuit of depth $O(\sqrt{s})$ computing the same function.*

In the following proof, we use the property of synchronous circuits that given any level $LV(i)$, f is a function of exactly those functions computed at the gates in $LV(i)$. This property allows us to do function composition in terms of two circuits. However, for layered circuits, inputs could be in the jth layer for $j < i$. Thus the property no longer holds for layered circuits that are not synchronous.

Note that the notation LV stands for "levels", while in the previous section we used L for "layers".

Proof. Let f be the function computed by C. (If C has more than one output, the proof can be applied by considering each output function separately, and combine the resulting small depth circuits.) Since C is synchronous, every level in C forms a cut. Furthermore, given any level $LV(i)$, f is a function of exactly those functions computed at the gates in $LV(i)$. We shall use this special property of synchronous circuits to compute f by the composition of two circuits.

Let $LV(0), LV(1), \ldots, LV(d)$ be the levels in C, where d is the depth of C, and $LV(0)$ contains the inputs. Let y be an integer whose value will be determined later. We say that a level $LV(i)$ is *small* if $|LV(i)| \leq y$ and *large* otherwise.

If C has many outputs, then it is possible that all the levels are large, but then the depth of C is at most $\frac{s}{y}$. Assume that C has at least one small level. Let $LV(k_0)$ be the small level farthest from the output of C.

Now let $g_1, \ldots, g_m$ be the gates in $LV(k_0)$, and let γ_i be the function computed at g_i. As noted above, f is a function of $\gamma_1, \ldots, \gamma_m$. Let $f = f'(\gamma_1, \ldots, \gamma_m)$. Then by Lemma 3.4.1, given $\gamma_1, \ldots, \gamma_m$ as inputs, f' can be computed by a synchronous circuit F of depth $O(|LV(k_0)|) = O(y)$.

Let C' be the multiple-output subcircuit of C with outputs $g_1, \ldots, g_m$. That is, C' consists of the levels $LV(0), \ldots, LV(k_0)$ in C, where $LV(k_0)$ contains the outputs of C'. (If $LV(k_0) = LV(0)$, then C' consists of only one level, formed by the inputs of C.) We use the outputs of C' as inputs for the circuit F. The resulting combined circuit F' is a synchronous circuit computing f. Note that all the levels $LV(0), \ldots, LV(k_0 - 1)$ are large. Since there are at most $\frac{s}{y}$ large levels in C, the depth of F' is at most $O(y + \frac{s}{y})$.

Thus, we obtain a synchronous circuit of depth at most $O(y + \frac{s}{y})$. Letting $y = \sqrt{s}$, we can simulate C by a synchronous circuit of depth $O(\sqrt{s})$. □

3.5 Two-Person Pebble Game on Circuits with Small Separators

For the one-person and two-person pebble games, we shall present a divide-and-conquer strategy that relates the separator size to the number of pebbles used in each game. The idea was implicitly used in the one-person pebble game [102] [83] and the two-person pebble game [34] [141]. Same idea was also applied on graph layout problems of VLSI [136].

Theorem 3.5.1. *Let $G = (V, E)$ be a DAG that has separators of size $h()$. Let $n = |V|$. Then to pebble any node of G in the one-person pebble game takes*

$$O\left(\sum_{i=0}^{\lceil \log n \rceil} h\left((\frac{2}{3})^i n\right)\right).$$

pebbles.

Proof. We define the pebbling strategy inductively. If G has only one node, then

pebble that node directly. Now let $v \in G$ be the node we want to pebble. The strategy has two phases. In the first phase, let H be the subDAG with v as sink and the subset of the source of G that v depends on topologically as H's source. Let S be the separator of H. Since G has separators of size $h()$, S is guaranteed to exist. We now pebble each node $w \in S$. We do this by applying the strategy recursively on the subDAG H' with w as its sink. The source of H', however, could be the source of H or those nodes in S that are already pebbled (or both). We shall choose nodes as the source of H', such that, every path from the source to the sink does not contain any node in S, except the source itself. Also, once a node S is pebbled, we immediately remove all pebbles except those on the separator, and continue pebbling the next node in the separator. So at the end of the first phase, every node in S is pebbled.

In the second phase, let H' be the subDAG of G with v as sink. For the source of H', we choose those nodes such that every path from the source of H' to v does not contain any node from S, except the source nodes themselves. We confine the pebble game within H' in this phase. Since the separator of G is pebbled, we are allowed to recursively apply the strategy in H' until v is pebbled. Notice that in both phases, we can apply the strategy recursively in the respective subDAG because G has separators of size $h()$.

Let $p(|V|)$ be the maximum number of pebbles needed to pebble any node of G. We have $p(1) = 1$ in the base case. For the first phase in the induction step, notice that to pebble the separator $S = \{w_1, \ldots, w_m\}$, the maximum number of pebbles on the graph is at most

$$\max\{p(|V_1|), 1 + p(|V_2|), 2 + p(|V_3|), \ldots, h(|V|) + p(|V_m|)\},$$

respectively, where V_i is the set of nodes of the ith subDAG considered in the first phase. So it takes at most $h(|V|) + p(\frac{2}{3}|V|)$ number of pebbles in the first phase.

For the second phase, the maximum number of pebbles on the graph includes the pebbled separator, and those pebbles needed to pebble v. So the maximum number of pebbles on the graph is at most $h(|V|) + p(\frac{2}{3}|V|)$. Together with the first phase, we obtain the following recursion,

$$p(1) = 1, p(|V|) \leq h(|V|) + p(\frac{2}{3}|V|).$$

Use the induction hypothesis to solve the equation. Then we have

$$p(|V|) = O\left(\sum_{i=0}^{\lceil \log |V| \rceil} h\left((\frac{2}{3})^i |V|\right)\right).$$

□

Theorem 3.5.2. *Let $C = (V, E)$ be a Boolean circuit such that C has separators of size $h()$, where $h(t) = o(t)$. Then C can be two-person pebbled in time*

$$O\left(\sum_{i=0}^{\lceil \log_{3/2} |V| \rceil} h\left((2/3)^i |V|\right)\right).$$

Proof. Let $p(|V|)$ be the number of pebbles required by the pebbler to win in the two-person pebble game on C. In each round, the pebbler will separate C into two subDAGs to see which subDAG the currently challenged node belongs to, and then recurse on that subDAG. Notice that in general, both components may contain several disjoint subcomponents.

We now define the strategy recursively. If C has only one node, then the pebbler wins immediately after the challenger's initial move. Now suppose that the challenger puts a challenge on a node $g \in C$. Then the pebbler places pebbles on the nodes of some appropriately chosen separator S of C_1, where C_1 is the unique maximal subcircuit of C with g as its output. Let $i \geq 1$. There are two cases in the $(i+1)$th round.

1. The challenger re-challenges g. Let S_i be the separator of C_i chosen in the ith round. Let C_{i+1} be the unique maximal subcircuit of C_i with g as its output, such that the inputs of C_{i+1} are some inputs of C_i upon which g depends, or some nodes in the separator S_i, and the underlying undirected graph of C_{i+1} is connected. Furthermore, we require that every path from the inputs of C_{i+1} to g does not contain any node in S_i. Then the pebbler applies this strategy recursively, and in the next round, looks for an appropriate separator of C_{i+1}.

2. The challenger puts a new challenge on a node $w \in S$. Let C_{i+1} be defined as above, but with w as its output. Then, as above, the pebbler applies this

strategy recursively.

Let C_{i1} and C_{i2} be the two subDAGs of C_i defined by the separator S_i. Note that C_{i1} and C_{i2} might be disconnected, but we defined C_{i+1} to be a connected subcircuit of either C_{i1} or C_{i2}. We claim that the pebbler will win after at most $O(\log |V|)$ rounds. To see this, notice that after the first round, the challenger can only challenge a node that has just been challenged, or a node in the separator. So the challenged node is restricted to either $C_{i1} \cup S_i$ or $C_{i2} \cup S_i$ that have sizes at most $(\frac{2}{3} + o(1))|C_i|$ because by assumption C has separators of size $h()$ and $h(t) = o(t)$. This also implies that the size of C_{i+1} is at most $(\frac{2}{3} + o(1))|C_i|$, and the game must terminate in at most $O(\log |V|)$ rounds.

For the number of pebbles used, we have the following recursion:

$$p(1) = 0, p(|C_i|) \leq h(|C_i|) + p\left(\left(\frac{2}{3} + o(1)\right)|C_i|\right).$$

Solving the above recursion yields

$$p(|V|) = O\left(\sum_{i=0}^{\lceil \log_{3/2} |V| \rceil} h\left((2/3)^i |V|\right)\right).$$

□

Here is the simplified version of the above two theorems.

Corollary 3.5.3. *Let $G = (V, E)$ be a DAG that has separators of size $h()$. Then the followings are true.*

1. *In the one-person pebble game, it takes at most $O(h(|V|) \log |V|)$ pebbles to pebble any node.*
2. *In the two-person pebble game, the pebbler can win the game in time $O(h(|V|) \log |V|)$.*

Proof. Note that $h(n)$ is non-increasing when n gets smaller, because we do not need larger separators to divide a smaller circuit. So we have

$$h\left((\frac{2}{3})^i |V|\right) \leq h(|V|) \text{ for all } i \geq 0.$$

Hence we have

$$O\left(\sum_{i=0}^{\lceil \log |V| \rceil} h\left((\frac{2}{3})^i |V|\right)\right) = O(h(|V|)\log |V|).$$

□

3.6 Simulation of Circuits with Small Separators

Theorem 3.5.2 gives the following corollary.

Corollary 3.6.1. *Let C be a bounded fanin circuit of size s and C has separators of size $h()$. Then we have following simulations.*

1. *There exists an equivalent tree-like bounded fanin circuit F of depth*

$$O\left(\sum_{i=0}^{\lceil \log s \rceil} h\left((\frac{2}{3})^i s\right)\right).$$

2. *There exists an equivalent tree-like unbounded fanin circuit F' of depth $O(\log s)$, and the fanin of F' is bounded by $2^{O(h(s))}$.*

3. *There exists an equivalent tree-like semi-unbounded fanin circuit F'' of depth $O(\log s \log h(s))$, and the fanin of F'' is bounded by $2^{O(h(s))}$.*

Proof. The first claim follows immediately from Theorem 3.5.2 and Theorem 2.2.2. Now notice that the maximum number of pebbles used in each round is $O(h(s))$, and the pebbler can win the game in $O(\log s)$ rounds. The second and the third claims then follow from Theorem 2.1.4. □

Given a Boolean function, we can construct a constant-depth, unbounded fanin circuit from its DNF representation. However, the fanin of the circuit is exponential in the input length. The first and the second claims show that the separator size not only reduces the depth, but the fanin as well.

Now we state several corollaries followed from the pebbling strategy based on separators. The first result is an alternative proof for Spira's theorem [129] (Theorem 1.2.3).

Proof. Since F is a tree and any weakly-connected subgraph of a tree is still a tree, F has a 1-separator by Theorem 2.4.5. The result follows from Corollary 3.6.1. □

Corollary 3.6.2. *Given a bounded fanin planar circuit C of size s, then we have the following simulations of C.*

1. *There exists a bounded fanin planar circuit F of depth $O(\sqrt{s})$ that computes the same function.*
2. *There exists an unbounded fanin planar circuit F of depth $O(\log s)$ and fanin $2^{O(\sqrt{s})}$ that computes the same function.*
3. *There exists a semi-unbounded fanin planar circuit F' of depth $O(log^2 s)$ and fanin $2^{O(\sqrt{s})}$ that computes the same function.*

Proof. Follows immediately from Theorem 2.4.6, Corollary 3.6.1, and the observation that any subgraph of a planar graph is still planar. □

This improves the previous known best result, that any planar circuit of size s can be simulated by a circuit of depth $s/\log s$.

Cook [25] showed that the pyramid graph needs $\Omega(\sqrt{s})$ pebbles in the one-person pebble game. Together with Theorem 2.1.5 and Corollary 3.6.2, the pyramid graph gives a tight $\theta(\sqrt{s})$ bound on the number of pebbles needed to win the two-person pebble game on planar graphs.

By *cylindrical circuits*, we refer to those circuits that can be embedded on the cylinder such that no two wires cross. Cylindrical circuits were studied by Hansen, Miltersen, and Vinay [56], and Limaye, Mahajan,and Sarma [80]. Since cylindrical and planar circuits can be transformed to each other by homeomorphisms, our previous results also apply to cylindrical circuits.

Corollary 3.6.3. *Given a cylindrical bounded fanin circuit C of size s, we have the following simulations of C.*

1. *There exists a bounded fanin tree-like circuit F of depth $O(\sqrt{s})$ that computes the same function.*
2. *There exists an unbounded fanin tree-like circuit F of depth $O(\log s)$ and fanin $2^{O(\sqrt{s})}$ that computes the same function.*

3. *There exists a semi-unbounded fanin tree-like circuit F' of depth $O(\log^2 s)$ and fanin $2^{O(\sqrt{s})}$ that computes the same function.*

From Theorem 2.4.7, we also have similar results for circuits with bounded genus.

Corollary 3.6.4. *Given a bounded fanin circuit C of size s and genus g, we have the following simulations of C.*

1. *There exists a bounded fanin tree-like circuit F of depth $O(\sqrt{gs})$ that computes the same function.*

2. *There exists an unbounded fanin tree-like circuit F of depth $O(\log s)$ and fanin $2^{O(\sqrt{gs})}$ that computes the same function.*

3. *There exists a semi-unbounded fanin tree-like circuit F' of depth $O(\log s(\log g + \log s))$ and fanin $2^{O(\sqrt{gs})}$ that computes the same function.*

If a graph G does not have K_k as its minor, neither can any subgraph of G. So G has separators of size $h()$, where $h(t) = O(k^{3/2}t^{1/2})$. From Theorem 2.4.8, we also have similar results for circuits with excluded-minors.

Corollary 3.6.5. *Given a bounded fanin circuit C of size s and has no K_h as a minor, we have the following simulations of C.*

1. *There exists a bounded fanin tree-like circuit F of depth $O(k^{\frac{3}{2}}s^{\frac{1}{2}})$ that computes the same function.*

2. *There exists an unbounded fanin tree-like circuit F of depth $O(\log s)$ and fanin $2^{O(k^{3/2}s^{1/2})}$ that computes the same function.*

3. *There exists a semi-unbounded fanin tree-like circuit F' of depth $O(\log s(\log k + \log s))$ and fanin $2^{O(k^{3/2}s^{1/2})}$ that computes the same function.*

Chapter 4

Generalization of Spira's Theorem

In Chapter 3, we showed how to simulate circuit size by circuit depth using the two-person pebble game. However, the circuit families generated by the pebble game is not uniform. In this chapter, we focus on the problem of generating uniform simulating circuit families. That is, we want to generate the circuit descriptions of the simulating circuits in small space. For circuits with constant-size segregators or separators, the descriptions of the simulating circuits can be generated in space $O(\log^2 s)$. We also note that our simulation works for any circuit, and if the circuit has a segregator of size $f(s)$, we obtain a simulating circuit of depth at most $O(f(s)\log s)$, whose circuit description can be generated in space $O(f(s)\log^2 s)$. Furthermore, the value $f(s)$ does not have to be provided in advance.

We also observe that a variant of our technique can be applied to monotone circuits. In particular, any monotone circuit of size s that has separator or segregator of size $f()$ can be simulated by a *monotone* circuit of depth $O(f(s)\log s)$, and the description of the simulating circuit can be generated in space $O(f(s)\log^2 s)$. This generalizes Wegener's result [145], which showed that Spira's theorem [129] (Theorem 1.2.3) is still true for monotone formulas.

One major advantage of a uniform simulating circuit family is that we can use it to solve the Circuit Value Problem in small space. See Chapter 5 for details.

Most of the results in this chapter can be found in Gál and Jang [46].

4.1 A Generalization of Spira's Theorem for Boolean Circuits with Small Segregators or Separators

By Lemma 2.4.4, if the circuit has separators of size $f()$, then it must also have segregators of size $f()$. Therefore in the following we will focus on circuits with small segregators.

We state and prove the following generalization of Spira's theorem for the basis $\{\wedge, \vee, \neg\}$ with fanin at most 2. It is easy to see that Theorem 4.1.1 can be generalized to Boolean circuits over arbitrary complete basis with bounded fanin, since any complete basis can implement the selector used in expression (4.1).

Theorem 4.1.1. *Any Boolean circuit of size s with segregators of size $f()$ can be simulated in depth $O(f(s))$ if $f(s) = \Omega(s^{\varepsilon})$ for some constant $\varepsilon > 0$, and in depth $O(f(s) \log s)$ otherwise.*

Proof. The construction is defined recursively. Let $U = \{u_1, \ldots, u_p\}$ be the segregator of C with size $p \leq f(s)$. Let $C_1, \ldots, C_p$ be the subcircuits of C corresponding to the nodes of the segregator, that is the node u_j is the output of the subcircuit C_j, for $j = 1, \ldots, p$. Let g_j be the Boolean function computed by C_j. Let v be the output node of the circuit C, and let $\hat{C}$ be the circuit with output node v, obtained from C by replacing the nodes in U by new variables $y_1, \ldots, y_p$. Thus, if the original circuit C has n variables, then $\hat{C}$ may have up to $p+n$ variables. It is possible that $\hat{C}$ has less than $p+n$ variables, if some of the original inputs get disconnected from the output v after removing the nodes of the segregator from the circuit.

We enumerate all Boolean vectors $c \in \{0,1\}^p$. Let $c_i = \langle c_{i,1}, c_{i,2}, \ldots, c_{i,p} \rangle$ be the ith Boolean vector of length p, for $i = 1, \ldots, 2^p$, according to some fixed ordering. Let $\hat{C}_i$ be the circuit obtained from $\hat{C}$ by fixing the values of the variables $y_1, \ldots, y_p$ to the bits $c_{i,1}, \ldots, c_{i,p}$, respectively. Let $h_i : \{0,1\}^n \to \{0,1\}$ be the Boolean function computed by the circuit $\hat{C}_i$.

Then, the Boolean function computed by the circuit C can be represented using the following expression:

$$\bigvee_{i=1}^{2^p} \left(h_i \wedge \bigwedge_{j=1}^{p} ((g_j \wedge c_{i,j}) \vee (\neg g_j \wedge \neg c_{i,j})) \right) \tag{4.1}$$

To see that expression (4.1) is indeed the function computed by C, note that on each input x, exactly one of the functions H_i evaluates to 1, where H_i is defined by

$$H_i = \bigwedge_{j=1}^{p} \left((g_j \wedge c_{i,j}) \vee (\neg g_j \wedge \neg c_{i,j}) \right).$$

For a given input $x \in \{0,1\}^n$, let $v_x = \langle g_1(x), g_2(x), \ldots, g_p(x) \rangle \in \{0,1\}^p$. Notice that $H_i(x)$ is nonzero if and only if $v_x = c_i$. Finally, note that $C(x) = h_i(x)$ when $v_x = c_i$.

Next we will represent the functions h_i for $i = 1, \ldots, 2^p$ and g_j for $j = 1, \ldots, p$. We could proceed with a straightforward recursion, if we could claim that each subcircuit $C_1, \ldots, C_p$ and each circuit $\hat{C}_i$ for $i = 1, \ldots, 2^p$ has size at most $2s/3$. In fact, we do know that every subDAG of the underlying DAG of C with the nodes of U removed has size at most $2s/3$. However, the output node of the subcircuit C_j is u_j, and u_j is a member of the segregator U. Note that the underlying DAGs of the circuits $\hat{C}_i$ are identical (they only differ from each other in the substituted constants), and their output node v is the output node of the "original" circuit C. The node v may or may not participate in the segregator. If the node v participates in the segregator, then the functions h_i are constants and the recursion stops.

We can compute the function g_j (computed at gate u_j) by an additional gate if we compute the functions computed at the two children of the gate u_j. If none of the children participates in the segregator, then we know that their subcircuits must have size at most $2s/3$. However, it is possible that children of segregator nodes are also included in the segregator. Let S_j be the set of nodes in the segregator, that are predecessors of u_j, such that there is a path from each of them to u_j that consists only of segregator nodes. We also include u_j in S_j. That is, S_j forms a subcircuit with output u_j that consists of segregator nodes. Let B_j be the "boundary" of S_j formed by nodes that are not in the segregator, that is, B_j contains the children of the nodes in S_j that are not included in the segregator. Then we can compute the function g_j from the functions computed at the nodes in B_j (these can be computed by subcircuits of size at most $2s/3$) with an additional set of gates corresponding to the segregator nodes in S_j. Since $|S_j| \leq p$, this takes additional depth at most p.

To summarize, we can compute the functions h_i and g_j, by first computing in parallel the functions corresponding to all subcircuits after removing the nodes of the

segregator. We know that each such subcircuit has size at most $2s/3$, and we can use our construction recursively on these smaller size circuits. Then we finish computing every function h_i and g_j we need, by adding the gates corresponding to the nodes participating in the segregator. This will take at most an additional $p \leq f(s)$ depth. Then we can compute the function computed by C by expression (4.1). This takes at most an additional $p + \lceil \log(p+1) \rceil + 3 = O(f(s))$ depth. Thus, in each iteration, we increase the depth by at most $O(f(s))$. Since the size is reduced by a constant factor in each iteration, we are done after $O(\log s)$ steps. More precisely, the depth of the final circuit is $O\left(\sum_{i=0}^{\lceil \log_{3/2} s \rceil} f\left((2/3)^i s\right)\right)$. Thus the depth of the final circuit is $O(f(s))$ if $f(s) = s^\epsilon$ for some constant $\epsilon > 0$, or $O(f(s) \log s)$ otherwise. □

Theorem 4.1.2. *The class of languages decided by non-uniform families of polynomial-size circuits with constant-size segregators equals non-uniform NC^1.*

Proof. Immediately follows from Theorem 4.1.1. □

Robertson and Seymour [116] showed that if a graph has treewidth k, then the graph also has separator size $O(k)$. Together with Lemma 2.4.4 and Theorem 4.1.2, a polynomial-size circuit with treewidth k can be simulated in depth $O(k \log n)$. This improves a result in [63], which showed that Boolean circuits of size $n^{O(1)}$ and treewidth k can be simulated in non-uniform depth $O(k^2 \log n)$. We refer interested readers to [32] and [38] for more background on treewidth.

4.1.1 A Generalization of Spira's Theorem for Monotone Circuits with Small Segregators or Separators

In this section we consider monotone circuits, i.e. circuits over the basis $\{\wedge, \vee\}$ with fanin 2. Wegener [145] proved Theorem 1.2.3 for monotone Boolean formulas. The following theorem generalizes his result to monotone circuits with small segregators.

Theorem 4.1.3. *Any monotone Boolean circuit C of size s with segregators of size $f()$ can be simulated by a monotone Boolean circuit in depth $O(f(s))$ if $f(s) = \Omega(s^\varepsilon)$ for some constant $\varepsilon > 0$, and in depth $O(f(s) \log s)$ otherwise.*

Proof. We first give a monotone version of expression (4.1), and the rest of the proof follows from the proof for Theorem 4.1. As in the previous section, we enumerate all Boolean vectors $c \in \{0,1\}^p$. Let $c_i = \langle c_{i,1}, c_{i,2}, \ldots, c_{i,p} \rangle$ be the ith Boolean vector

of length p, for $i = 1, \ldots, 2^p$, according to some fixed ordering. We assume that the c_1 is the all-zero vector. Let $\hat{C}_i$ be the circuit obtained from $\hat{C}$ by fixing the values of the variables $y_1, \ldots, y_p$ to the bits $c_{i,1}, \ldots, c_{i,p}$, respectively, where $\hat{C}$ and $y_1, \ldots, y_p$ are defined as in the proof of Theorem 4.1.1. Let $h_i : \{0,1\}^n \to \{0,1\}$ be the Boolean function computed by the circuit $\hat{C}_i$. Let G_i be defined as follows for $i \geq 2$:

$$G_i = \bigwedge_{\substack{1 \leq j \leq p \\ c_{i,j}=1}} g_j.$$

We claim that the function computed by C can be represented by the following expression:

$$h_1 \vee \bigvee_{i=2}^{2^p} (h_i \wedge G_i) = h_1 \vee \bigvee_{i=2}^{2^p} \left(h_i \wedge \bigwedge_{\substack{1 \leq j \leq p \\ c_{i,j}=1}} g_j \right). \tag{4.2}$$

To prove this claim, first we define a relation $<$ on Boolean vectors of length p: given $c_u = \langle c_{u,1}, c_{u,2}, \ldots, c_{u,p} \rangle$ and $c_v = \langle c_{v,1}, c_{v,2}, \ldots, c_{v,p} \rangle$, $c_u < c_v$ iff $c_{u,k} \leq c_{v,k}$ for all $1 \leq k \leq p$, and $c_{u,r} < c_{v,r}$ for some $1 \leq r \leq p$.

Let $w_x = \langle g_1(x), g_2(x), \ldots, g_p(x) \rangle \in \{0,1\}^p$. Notice that if $w_x = c_i$, then $G_i(x) = 1$. On the other hand, for any vector c_k such that $w_x < c_k$ or w_x is not comparable to c_k, $G_k(x) = 0$. This can be proved by the following argument. If $c_i < c_k$, then there exists an l such that $c_{i,l} < c_{k,l}$. Then $g_l(x) = c_{i,l} = 0$, which implies that $G_k(x) = 0$. If c_i and c_k are not comparable, then there must also exist an l such that $c_{i,l} = 0$ but $c_{k,l} = 1$. Thus $g_l(x) = 0$ and $G_k(x) = 0$. This implies that on a given $x \in \{0,1\}^n$, expression (4.2) evaluates to

$$h_1(x) \vee \bigvee_{\substack{2 \leq k \leq 2^p \\ c_k < w_x}} (h_k(x) \wedge G_k(x)).$$

Since C is monotone, $c_k < c_i$ implies $h_k(z) \leq h_i(z)$ for all $z \in \{0,1\}^n$. Note

also that since $w_x = c_i$, $G_k(x) \leq G_i(x) = 1$ for any k. Therefore we have

$$\begin{aligned} & h_1(x) \vee \bigvee_{\substack{2 \leq k \leq 2^p \\ c_k < c_i}} (h_k(x) \wedge G_k(x)) \\ = \; & h_1(x) \vee (h_i(x) \wedge G_i(x)) \\ = \; & h_1(x) \vee h_i(x) \\ = \; & h_i(x) \text{ since } c_1 < c_i \end{aligned} \tag{4.3}$$

Recall that $C(x) = h_i(x)$ when $w_x = c_i$.

□

4.2 Finding Minimum Size Segregators in Small Space

4.2.1 Segregators of Directed Acyclic Graphs

In this section, we give a space-efficient algorithm to find a minimum size segregator in arbitrary directed acyclic graphs.

We will use the following space-efficient algorithm for reachability in directed graphs by Savitch [123], to count the number of predecessors of a given node.

Theorem 4.2.1. *[123] Given a directed graph G on s nodes and two nodes $u, v \in G$, there exists a deterministic Turing machine that decides if there is a path from u to v in G using space $O(\log^2 s)$.*

Lemma 4.2.2. *Let G be a DAG with s nodes. There exists a deterministic Turing machine M such that, on input G, if G has a segregator of size $f(s)$, then M outputs a segregator of G of size at most $f(s)$ using space $O(f(s) \log s + \log^2 s)$.*

Proof. We first define a Turing machine M_1 that takes G and a node $v \in G$ as input, and computes the number of predecessors of v in G, i.e. the number of nodes u such that there exists a directed path from u to v in G. In the beginning M_1 initializes a counter to 1. Then M_1 uses Theorem 4.2.1 to check, one-by-one, for each node $u \in G \setminus \{v\}$ if there is a directed path from u to v in G. For each node $u \in G \setminus \{v\}$ such that v is reachable from u, the counter is incremented. The space used to check the reachability of v from u is reused when checking for reachability

from the next node in $G \setminus \{v\}$. Thus M_1 uses $O(\log^2 s)$ space and computes the size of the subDAG with v as the root.

We now define M in Lemma 4.2.2 as follows. First M enumerates integers k such that $1 \leq k \leq s$ in increasing order. For a fixed k, M enumerates subsets W of size k of the nodes in G in lexicographic order. For a given W, for every node $u \in G \setminus W$, let $G(u)$ denote the set of predecessors of u in $G \setminus W$. That is, $G(u)$ is the subDAG in $G \setminus W$ with u as its root. M uses M_1 to compute $|G(u)|$. If there exists one node $u \in G \setminus W$ such that $|G(u)| > \frac{2}{3}s$, then M continues to the next W, or the next k if every W of the current size has been already checked. Also, every time before continuing to the next W or the next k, M clears unnecessary information from the work tape.

We now argue that M will find a segregator of the smallest size. Observe that the set of nodes of G is a segregator of size s, so M is guaranteed to find a segregator. Since we try every k in increasing order, and we check for every subset W of size k whether or not it is a segregator, it is guaranteed that we will find a segregator of the smallest possible size in G.

We now argue that M only uses $O(f(s)\log s + \log^2 s)$ space. The description of G can be read using a counter of size $O(\log s)$. The enumeration and the storing of W both take $O(k \log s) = O(f(s) \log s)$ space. The computation of $|G(u)|$ takes $O(\log^2 s)$ space since M_1 uses $O(\log^2)$ space. Thus the space complexity to find a segregator of smallest size is $O(f(s)\log s + \log^2 s)$. □

Note that in the proof for Lemma 4.2.2, the input of M consists of only the description of the graph. M does not know the value of $f(s)$ in advance. Also, by Lemma 2.4.4, for graphs with separators of size k, the algorithm in Lemma 4.2.2 will also find a segregator of size at most k.

4.2.2 Segregators of Uniform Circuits

Intuitively, Lemma 4.2.2 seems to apply directly to circuits since circuits are also DAGs. However, the input of the Turing machine that has to generate the circuit C_n for a uniform family of circuits, is the unary representation of n (1^n), so the graph of the circuit C_n is not available directly. Since we want to generate the segregator using small space, we cannot store the description of C_n on the work tapes. As it is standard in such situations, we will generate the description of C_n as needed for

the machine in the proof of Lemma 4.2.2, but never store the complete description. We then have the following lemma.

Lemma 4.2.3. *Let $\mathcal{C}$ be a $h(n)$-space uniform family of circuits. Let $C_n \in \mathcal{C}$ be the Boolean circuit in the family with n inputs, and assume that C_n has size $s = s(n)$ and a segregator of size $f(s)$. Then there exists a deterministic Turing machine $\hat{M}$ that on input 1^n, outputs a segregator of C_n of size at most $f(s)$ using space $O(h(n) + f(s)\log s + \log^2 s)$.*

As in the case for directed graphs, for circuits with separators of size $f(s)$, the algorithm in Lemma 4.2.3 will also find a segregator of size at most $f(s)$.

4.3 Making the Generalization of Spira's Theorem Uniform: Generating the Simulating Circuits in Small Space

Let v be any node and Z be any set of nodes in the underlying graph of a circuit C_n. We denote by $C_{v,Z}$ the circuit obtained from the subcircuit C_v of C_n with output v by replacing every node in Z that participates in C_v by a new input variable.

Lemma 4.3.1. *Let $\mathcal{C}$ be a $h(n)$-space uniform family of circuits. Let $C_n \in \mathcal{C}$ be the circuit with n inputs in the family, and assume that C_n has size $s = s(n)$. Let v be any node and Z be any set of nodes in the underlying graph of C_n. Then there exists a Turing machine M_2 such that on input 1^n, v and Z, M_2 outputs the description of the circuit $C_{v,Z}$. Furthermore, M_2 runs in space $O(h(n) + \log^2 s)$.*

Note that if $Z = \emptyset$, or if Z does not contain any predecessors of v then $C_{v,Z}$ is simply the subcircuit C_v. Similarly to the circuit $\hat{C}$ in the proof of Theorem 4.1.1, if the size of Z is r, and C_v depends on n' input variables, then $C_{v,Z}$ may have up to $n' + r$ variables. If $v \in Z$, then $C_{v,Z}$ is simply a new variable.

Proof. Let M_1 be the Turing machine that on input 1^n generates the description of C_n using space $O(h(n))$. M_2 will use M_1 to generate information about the circuit C_n as needed. As before, the full description of C_n will never be stored. M_2 will use Theorem 4.2.1 to check if a given node is part of the subcircuit C_v, using space $O(\log^2 s)$. As before, space can be reused when checking for a new node. Note that

the size of the set Z can be larger than the size of the subcircuit C_v, but it will be at most s, which is the size of the whole circuit C_n. M_2 will need to work with a counter of size $\log |Z|$, but $\log |Z| < \log^2 s$. □

Lemma 4.3.2. *Let $\mathcal{C}$ be a $h(n)$-space uniform family of circuits. Let $C_n \in \mathcal{C}$ be the circuit with n inputs in the family, and assume that C_n has size $s = s(n)$. Let v be any node and Z be any set of nodes in the underlying graph of C_n. Also assume that C_n has segregators of size $f()$. Then there exists a Turing machine M_3 such that on input 1^n, v and Z, M_3 outputs a minimum size segregator of $C_{v,Z}$ using space $O(h(n) + f(s) \log s + \log^2 s)$.*

Proof. Let M_2 be the Turing machine in Lemma 4.3.1 that generates the description of $C_{v,Z}$ in space $O(h(n) + \log^2 s)$. Let M be the Turing machine in the statement of Lemma 4.2.2, that takes a directed graph G as input, and outputs a minimum size segregator of G. The machine M_3 will simulate M on the underlying directed graph of $C_{v,Z}$. However, as before, the full description of the graph will never be stored. Instead, whenever M_3 needs some information about the graph, it lets M_2 run, (without recording its output), until the required information is generated. The size of the subcircuit $C_{v,Z}$ is $s' \leq s$. Since C_n has segregators of size $f()$, we know that $C_{v,Z}$ has a segregator of size $f(s')$. Recall that M always finds a minimum size segregator, thus it will find a segregator of size $f(s') \leq f(s)$. Since M runs in space $O(f(s) \log s + \log^2 s)$, the total space used will be $O(h(n) + f(s) \log s + \log^2 s)$. □

Now we are ready to prove a uniform version of Theorem 4.1.1.

Theorem 4.3.3. *Let $\mathcal{C}$ be an $h(n)$-space uniform family of Boolean circuits. Let $C_n \in \mathcal{C}$ be the Boolean circuit on n inputs with size $s = s(n)$. Suppose that C_n has segregators of size $f()$. Let $g(s) = f(s)$ if $f(s) = \Omega(s^c)$ for some constant $c > 0$ and $f(s) \log s$ otherwise. Then $\mathcal{C}$ can be simulated by a $O(h(n) + g(s) \log s)$-space uniform family of Boolean circuits of depth $O(g(s))$.*

Proof. We show that the construction in the proof of Theorem 4.1.1 can be generated by a machine M^* within the appropriate space bounds. M^* on input 1^n will output the description of the depth $O(g(s))$ circuit simulating the circuit $C_n \in \mathcal{C}$.

M^* generates the simulating circuit essentially as described in the proof of Theorem 4.1.1. In each step of the recursion, M^* has to do the following:

1. Find a segregator S of the current subcircuit, and store the list of nodes of S in workspace.

2. Find and store the list of nodes that participate in $B = \cup_{j=1}^{|S|} B_j$. Note that a given node may belong to B_j for more than one j, but $|\cup_{j=1}^{|S|} B_j| \leq 2|S|$, since B_j contains only children of segregator nodes. Thus, if $|S| = p$, it takes $O(p \log s)$ space to store the list of nodes in B. We can generate this list using $\hat{M}_1$, where $\hat{M}_1$ is the Turing machine that on input 1^n generates the description of C_n using space $O(h(n))$. We will run $\hat{M}_1$ several times, reusing space, and never store the full description of the circuit, as discussed before. For finding the set B_j, we have to find the set S_j and store it until we are finished generating B_j. For each j this takes $O(p \log s)$ workspace. We reuse this space when we move on to the next j. For each node of B_j that we find, we check if we have already added it to the list, so the full list B takes at most $O(p \log s)$ workspace to store.

3. Output the description of the part of the circuit that corresponds to the current subcircuit. This is based on the expression (4.1), and the sets B_j and S_j. We produce the description of the part of the circuit to compute g_j, while we have B_j and S_j stored in memory. We reuse space when we move on to the next j. Recall that the output is not part of the space bound. (We do keep S and the full list B until the end of processing the subcircuit, and maybe longer as we see below.)

The recursion will continue to process the subcircuits $\hat{C}_i$ (functions h_i) defined in the proof of Theorem 4.1.1, and the subcircuits of the nodes in B. Recall that each of these subcircuits has size at most $2/3$ of the last subcircuit. The recursion stops when a subcircuit is either constant or an input variable. We need a counter of size p to enumerate the Boolean vectors substituted, and to enumerate the functions h_i, for $i = 1, \ldots, 2^p$.

We reuse space as we proceed to the next recursive step. However, to be able to proceed with the recursion, we need to retain some information about the segregators S, the sets B and list of values substituted for segregator nodes from previous recursive steps to be able to generate and process the current subcircuits. We process the subcircuits similarly to a depth first search in the recursion tree,

starting with the subcircuits corresponding to the set B and leaving the subcircuit for the functions h_i for last. Recall that there is only one subcircuit to consider for the functions h_i, they just differ in the values of constants substituted.

We keep S, B and list of values substituted for nodes in S from previous steps along the current path in the recursion tree. Since there are $\log s$ stages of the recursion, at any point we keep at most $\log s$ segregators with their corresponding set B and list of values. This takes $O(\sum_{i=1}^{\log s} f(s/2^i) \log s) = O(g(s) \log s)$ space.

At the first iteration, we simply use the machine $\hat{M}$ from Lemma 4.2.3 to find a segregator. Now we describe how to find a segregator of the current subcircuit during the recursion. To find a segregator for the subcircuits with outputs in the sets B described above, we use M_3 with input 1^n, u where u is the output of the subcircuit, and $Z = \emptyset$. (For processing the subcircuits corresponding to nodes in the sets B we do not need to worry about the segregators that we stored from previous levels of the recursion.) For the subcircuits $\hat{C}_i$ (functions h_i) we use M_3 with input 1^n, v, where v is the output node of the subcircuits $\hat{C}_i$ (recall that they have the same output node, they only differ in the constants substituted), and Z where Z is the union of all the segregators currently stored.

In each step of the recursion, M_3 finds the current segregator in at most $h(n) + O(\log^2 s + f(s) \log s)$ space by Lemma 4.3.2. Note that after each invocation of Lemma 4.3.2, its workspace can be reused.

Thus on input 1^n, the space used to construct the new circuit is at most $O(h(n) + \log^2 s + g(s) \log s) = O(h(n) + g(s) \log s)$ since $g(s) = \Omega(\log s)$. □

Chapter 5

Circuit Value Problem

In Chapter 4, we gave uniform simulating circuit families for circuits with small separators or segregators. That is, we give a space-efficient algorithm to generate the circuit description of the simulating circuit families. One application is to solve the Circuit Value Problem in small space. The Boolean Circuit Value Problem is defined as follows: given the description of a circuit C and an assignment x to the variables of C, compute the value of the output of the circuit C evaluated on the assignment x. It is not known if the general Circuit Value Problem, which is P-complete, can be solved in $o(n/\log n)$ space. In this chapter, we consider the Circuit Value Problem for three circuit families: planar, layered, and synchronous circuits. It is known that these variants of the Circuit Value Problem are all P-complete. See [53] and [54].

First we solve the Planar Circuit Value Problem in $O(\sqrt{n}\log n)$ space using Theorem 4.3.3. Then we show that the Layered Circuit Value Problem and the Synchronous Circuit Value Problem can be solved in $O(\sqrt{n})$ space. For layered circuits and synchronous circuits, we use different techniques to solve the Circuit Value Problem, since we no longer have uniform simulating circuit families for them. See Section 5.2 for details.

5.1 CVP for Planar Circuits is in $SPACE(\sqrt{n}\log n)$

As an application of Theorem 4.3.3, we obtain a bound on the space complexity of the Circuit Value Problem for Boolean circuits with small segregators (or separa-

tors).

Theorem 5.1.1. *The Boolean Circuit Value problem for circuits that have size s and segregators (or separators) of size $f(s)$ is in $SPACE(f(s)\log s)$ if $f(s) = \Omega(s^{\varepsilon})$ for some constant $\varepsilon > 0$, and $SPACE(f(s)\log^2 s)$ otherwise.*

Proof. Let $g(s) = f(s)$ if $f(s) = \Omega(s^{\varepsilon})$ for some constant $\varepsilon > 0$, and $g(s) = f(s)\log s$ otherwise. Since the description of C is given in the input, by the proof of Theorem 4.3.3, using $O(g(s)\log s)$ space, we can generate a circuit C' of depth $O(g(s))$ that simulates C. Then we can evaluate C' in the given assignment using the argument of Theorem 1.2.1 using space $O(g(s))$. □

Theorem 5.1.1 immediately implies the following theorem.

Theorem 5.1.2. *The Boolean Circuit Value problem for circuits with constant-size segregators (or separators) is in $SPACE(\log^2 n)$.*

Theorem 5.1.3. *The Planar Circuit Value Problem is in $SPACE(\sqrt{n}\log n)$.*

Proof. Immediately follows from Theorem 2.4.6 and Theorem 5.1.1. □

5.2 CVP for Layered Circuits and Synchronous Circuits is in $SPACE(\sqrt{n})$

In the following we consider the Layered Circuit Value Problem and the Synchronous Circuit Value Problem. We first show that the Layered Circuit Value Problem can be solved in $O(\sqrt{s})$ space. By Lemma 3.1.7, every synchronous circuit is layered. It then follows that the Synchronous Circuit Value Problem can be also solved in $O(\sqrt{s})$ space.

Recall the definitions of height, layer, and layered circuits in Section 3.1. In the followings we assume that a layered circuit has one output unless stated otherwise.

Lemma 5.2.1. *Given the description of a layered Boolean circuit C of size s, and any gate g in C, the height of g can be computed in $O(\log s)$ space.*

Proof. We define the following Turing machine M. M follows a path starting from g until it reaches the output of C. This can be done by scanning the description of

C to see which gates in C have g as a child. If the gate g has fanout more than 1, M chooses arbitrarily which parent of g to visit next, say for example the first such gate in the description. Once a parent of g is found, say h, we let h be the current gate and repeat the above process. M also counts the number of edges in the path and outputs that number when the output gate is reached. It is easy to see that M computes the height of g, since all paths from the gate g to the output of C have the same length. M uses $O(\log s)$ space since it only needs to remember the name of the current gate along the path and a constant number of counters using $O(\log s)$ space for each. □

Lemma 5.2.2. *Let C be a layered Boolean circuit of size s and height h. Let $0 \leq i < j \leq h$ be two integers. Let g be any gate in the ith layer, and let $C(g, j)$ be the subcircuit of C whose output is g, and whose inputs are all the gates in the jth layer that are connected to g by a path to g, and those circuit inputs that have a parent gate in layers $i, \ldots, j-1$. Then given the description of C, g, and j, the description of $C(g, j)$ can be computed in $O(\log^2 s)$ space.*

Proof. We now define a Turing machine M^*. We are going to produce a description of $C(g, j)$ according to Definition 1.1.2. First we list those circuit inputs that are participating in the subcircuit. Then we list the names of those gates u in the jth layer that serve as inputs to $C(g, j)$. We use Lemma 5.2.1 to check if a given gate is in the jth layer. We can use Savitch's theorem [123] (Theorem 4.2.1) to test if a given circuit input or a given gate u is connected to g by a path.

Next for $k = j - 1$ to 0, M^* generates the quadruples of the gates in the kth layer. For a given k, M^* scans the description of C, and for each gate $v \in C$, first check if v is in the kth layer and then check if v is connected to g. If yes, then M^* writes the quadruple corresponding to gate v to the output tape. It is clear that M^* uses $O(\log^2 s)$ space. □

Notice that the above algorithm can be easily modified such that it outputs the description of the subcircuit in any pre-determined format of circuit description. Also observe that the space used is dominated by the space of testing directed connectivity.

Theorem 5.2.3. *Given a description of a layered Boolean circuit C of size s, and an input assignment x to C, there exists a Turing machine that evaluates C on x*

using $O(\sqrt{s})$ space. That is, the Layered Circuit Value Problem can be solved in $O(\sqrt{s})$ space.

The idea is to evaluate the circuit layer-by-layer if the layers are small, and use Borodin's theorem [11] (Theorem 1.2.1) when the layers are large. Since there cannot be too many large layers, we can bound the space. We now give the complete proof.

Proof. Let $y > 1$ be some positive number. We call a layer in C *large* if the number of gates in the layer is greater than y, or *small* otherwise. We will determine the value of y later.

We define a Turing machine M as follows. In the beginning M computes the height of C. This can be easily done by applying Lemma 5.2.1. Given any $1 \leq j \leq h$, we can compute the number of gates in the jth layer also using Lemma 5.2.1.

Let h be the height of C. First we consider the hth layer. Note that the inputs of all gates in the hth layer are circuit inputs. If the hth layer is small, then M writes the values of the gates in the hth layer on the work tape. These values can be computed from the circuit description and x. If the hth layer is large, then M looks for the largest $i < h$ such that the ith layer is small. Note that this means the depth of every gate in the ith layer is at most s/y, since every layer between the ith layer and the hth layer is large. Similarly to the proof of Lemma 4.3.1, given the circuit description of C, we can produce the description of the subcircuit C_v for each gate v in the ith layer. Note however, we do not have enough space to store the circuit description of C_v. Instead we will produce the necessary information about each gate as needed. We use Theorem 1.2.1 to evaluate C_v, and we write the value of each gate in the ith layer on the work tape. Furthermore, M writes down values of gates according to their order in the circuit description of C. In this list, we do not store the names of the gates.

Let m be the largest m such that the mth layer is small. Then after what we described above, we have all the values of the gates in the mth layer written on the work tape.

Next let $j = m$. If $j = 0$, then we are done. Let the kth layer ($0 \leq k < j$) be the next small layer. That is, either $k = j - 1$, or all the layers between the kth and jth layers are large. For a given gate g in the kth layer, we will use Lemma 5.2.2 to generate the description of the circuit $C(g, j)$. Note that the inputs to $C(g, j)$ are

either circuit inputs, or gates in the jth layer. Also at this point, we have the values of all gates in the jth layer written on the work tape. We need the following claim.

Claim. Given the name of a gate u in the jth layer, we can compute in $O(\log s)$ space the index t such that u is the tth gate within the jth layer, according to the order of the gates in the circuit description of C.

Proof for the claim We enumerate the gates of the jth layer as follows. We consider each gate g in C according to their order in the circuit description of C. For each gate g, we use Lemma 5.2.1 to compute its height. If the height of g is j, we increment a counter. We stop when we reach u in the circuit description. The index t of u within the jth layer is the current value of the counter plus 1. It follows from Lemma 5.2.1 that the space used is $O(\log s)$. □

Putting it all together, M can produce the values of the gates in the kth layer as follows: M scans the description of C. For each gate g of C, we check (as before) if g is in the kth layer. If yes, we use Lemma 5.2.2 to produce the description of $C(g, j)$. Note that the depth of $C(g, j)$ is at most s/y (since all the layers between the kth layer and the jth layer are large). Then we use Theorem 1.2.1 to evaluate $C(g, j)$. However, we never actually store the description of $C(g, j)$. Instead, each time we need information about a given gate, we run the machine M^* from Lemma 5.2.2 (without actually recording its output) until we get the necessary information.

Recall that each input to $C(g, j)$ is either a circuit input (thus its value can be obtained from x) or a gate from the jth layer. Given the name of a gate u in the jth layer that is an input to $C(g, j)$, we can use the above claim to find its value among the values of the gates in the jth layer.

We keep the values of the jth layer on the work tape until we finish computing all the values in the kth layer. Then we let $j = k$ and we repeat the above process re-using space no longer needed.

M continues the above process until $j = 0$. Then M outputs the value of the output gate. The space used by M is bounded by $O(y + s/y + \log^2 s) = O(y + s/y)$. Let $y = \sqrt{s}$, then M uses $O(\sqrt{s})$ space. □

Since every synchronous circuit is layered, the Synchronous Circuit Value Problem can be solved using $O(\sqrt{s})$ space.

Chapter 6

The Degree of Boolean Circuits

In this chapter, we consider the notion of *degree* of Boolean circuits given by Definition 6.1.1. The degree of arithmetic circuits was well studied. See Section 2.5 for a brief review. The degree of Boolean circuits was first introduced by Skyum and Valiant [128] as a measure of parallelism. Our goal in this chapter is to establish some basic facts about the degree of Boolean circuits regarding simulation in small circuit depth. It is easy to see that Theorem 2.5.3 by Valiant, Skyum, Berkowitz and Rackoff [137] holds for Boolean circuits. In this chapter we consider two special circuit families where their degrees can be bounded. We observe some implications of Theorem 2.5.3 to these classes.

6.1 The Degree of Boolean Circuits

The degree of an arithmetic circuit was defined in Definition 2.5.1. As far as we know, Skyum and Valiant [128] were the first to study the degree of Boolean circuits. The definition is given as follows.

Definition 6.1.1. [128] Given a Boolean circuit C over basis $\{\wedge, \vee \neg\}$ with negations only at the inputs, the *degree* of each node $v \in C$, denoted as $\deg(v)$, is defined as follows. If v is a constant, then $\deg(v) = 0$. If v is a variable or a negated variable, then $\deg(v) = 1$. If v is an $\wedge$ gate, then $\deg(v) = \deg(w_1) + \deg(w_2)$, where w_1 and w_2 are the inputs of v. If v is an $\vee$ gate, then $\deg(v) = \max(\deg(w_1), \deg(w_2))$, where w_1 and w_2 are the inputs of v. The degree of C, $\deg(C)$, is the maximum degree over all gates in C.

Note that the above definition is meaningful only when the negations are in the input level. This restriction does not put a severe constraint on the circuit model by the following lemma, which can be easily proven using De Morgan's law. See Wegener [146] for a complete proof.

Lemma 6.1.2. *[146] Let C be a Boolean circuit of size s taking n inputs. Then there exists a Boolean circuit C' of size at most $2s+n$ computing the same function as C, and all negation gates in C' only negate the inputs of C'.*

Observe that in any Boolean circuit with negations only at the inputs, the degree of a gate is monotonically increasing along the path from any input to the output. Thus in any Boolean circuit with negations only at the inputs, the degree of the circuit is equal to the degree of the output gate.

Here we give another characterization of the degree of Boolean circuits. For a given circuit C, we define the *max-tree* of C as follows.

Definition 6.1.3. Given a Boolean circuit C with negations only at the inputs, the *max-tree* T of C is defined inductively as follows.

- T includes the output gate of C,
- For any $\wedge$ gate $g \in T$, let g_1 and g_2 be the immediate children of g in C. Let T_1 and T_2 be the max-tree of the subcircuits of C whose outputs are g_1 and g_2, respectively. Then we add two wires from the outputs of T_1 and T_2 to g and include *both* T_1 and T_2 in T.
- For any $\vee$ gate $g \in T$, let g_1 and g_2 be the immediate children of g in C. Let T_1 and T_2 be the max-tree of the subcircuits of C whose outputs are g_1 and g_2, respectively. Without loss of generality, assume that g_1 has larger degree than g_2 in C. Then we add one wire from the output of T_1 to g and include *only* T_1 in T. Break ties arbitrarily if g_1 and g_2 have the same degree in C.

Notice that for an $\wedge$ gate, we construct the max-tree such that the two subtrees under its two immediate children are disjoint. Also, given a gate $g \in C$, there might be many duplicates of g in the max-tree of C. It is also possible that the max-tree of C does not contain g at all, depending on the location of g in C.

The significance of the max-tree is that the number of leaves in the tree *equals* the degree of the circuit, as shown in the following theorem.

Theorem 6.1.4. *Let C be a Boolean circuit with negations only at the inputs. Then the degree of C is equal to the number of leaves in the max-tree of C.*

Proof. We shall prove by induction on the depth of C. Suppose C has depth 1 with g as the output gate of C. If g is an $\wedge$ gate, then we have $\deg(C) = 2$ and the number of leaves in the max-tree of C is also 2. If g is an $\vee$ gate, then we have $\deg(C) = 1$ and the number of leaves in the max-tree of C is also 1. So the claim is true in both base cases.

For the induction step, let g be the output gate of C taking inputs from g_1 and g_2. We then have two cases.

1. g is an $\vee$ gate. Let $h \in \{g_1, g_2\}$ such that $\deg(h) = \max(\deg(g_1), \deg(g_2))$ if $\deg(g_1) \neq \deg(g_2)$, and $h = g_1$ otherwise. Then we have $\deg(g) = \deg(h)$, and the max-tree of C is the same as g attaching to the root of the max-tree of the subcircuit with h as the output. By induction hypothesis, $\deg(h)$ is equal to the number of leaves of the max-tree of the subcircuit with h as the output. So the claim is true in this case.

2. g is an $\wedge$ gate. Then we have $\deg(g) = \deg(g_1) + \deg(g_2)$. Let $i \in \{1, 2\}$. By induction hypothesis, $\deg(g_i)$ is the same as the number of leaves of the max-tree of the subcircuit with g_i as the root. Since the max-trees of g_1 and g_2 are disjoint (i.e. do not share vertices), the claim is also true in this case.

□

An important observation is that in a tree, the number of leaves can be much smaller than the size of the tree, e.g. paths where the number of leaves is a constant. So in the divide-and-conquer method, we are more interested in how to reduce the number of *leaves* for max-trees. This is different from general trees, where we are more interested in reducing the total number of nodes.

From Theorem 2.5.3 by Valiant, Skyum, Berkowitz and Rackoff [137], we immediately have the following corresponding result for Boolean circuits.

Corollary 6.1.5. *(Implied by Theorem 2.5.3 and Definition 6.1.1) Let C be a Boolean circuit with negations only at the inputs such that C has size s and degree d computing f. Then there exists a Boolean circuit C' such that C' has size $O(s^3)$ and depth $O(\log s \log d)$ computing f.*

Proof. We construct an arithmetic circuit C_1 by replacing each AND gate by a $\times$ gate, an OR gate by a $+$ gate, and each occurrence of the input variables (negated or not) by a new indeterminate, e.g. x by x_1 and $\neg x$ by x_2. Furthermore, the degree of the polynomial computed by the output gate is the maximum degree over all gates. After applying the algorithm in Theorem 2.5.3, we switch back the gate types and inputs, e.g. x_1 to x and x_2 to $\neg x$. The claim then follows. □

Notice that C' constructed above is a layered circuit.

6.2 Skew Boolean Circuits

In general we only have trivial upper bound for the number of AND gates, i.e. the size of the circuit. However, if we put some restrictions on the inputs of the AND gates, then we may obtain nontrivial upper bound for the degree of the circuit. This is illustrated in the following lemma, which says that the degree of a skew Boolean circuit with negations only at the inputs is at most one plus the circuit size. A *skew Boolean circuit* is a Boolean circuit with a restriction on the AND gate: all its children except (possibly) one must be circuit inputs. In other words, every AND gate has at least one input variable.

In the following we briefly review known results and generalizations of skew circuits. Venkateswaran [140] first explicitly defined skew circuits to study the nondeterministic complexity classes. In particular, he showed that NL equals the class of languages computable by uniform families of polynomial-size skew circuits. Similar constructions were given by Masek [92] to study the space complexity and branching programs. See Razborov [114] for a survey in this area. Toda [131] generalized skew circuits and defined weakly skew circuits to study arithmetic circuits computing the determinant. A circuit C is *weakly skew* if for every $\wedge$ gate g with incoming wires (g_1, g) and (g_2, g), removing one of (g_1, g) or (g_2, g) will disconnect the subcircuit under g_1 or g_2 from C, respectively. Later Malod and Portier [89] gave another generalization of skew circuits to multiplicative-disjoint circuits. A circuit is *multiplicative-disjoint* if for every $\wedge$ gate g with incoming wires (g_1, g) and (g_2, g), the two subcircuits under g_1 and g_2 are disjoint. Notice that skew circuits is a subset of weakly skew circuits, and weakly skew circuits is a subset of multiplicative-disjoint circuits. Malod and Portier [89] further showed

that $LOGCFL$ equals uniform multiplicative-disjoint Boolean circuits with polynomial size. The multiplicative-disjoint circuits were also studied later by Mahajan and Rao [88]. Note that there are other interesting characterizations of $LOGCFL$. Venkateswaran [139] gave characterizations of $LOGCFL$ on three different models. McKenzie, Reinhardt, and Vinay [96] used circuits of multiplex select gates to characterize NC^1, L, $LOGDCFL$, and $LOGCFL$.

Lemma 6.2.1. *Let C be any skew Boolean circuit of size s with negations only at the inputs. Then the degree of C is at most $s+1$.*

Proof. We shall prove by induction on the size s. Suppose that C has only one gate. Then the degree of C is either 0, 1, or 2. So the claim is true in the base case.

Now suppose that the claim is true for all skew Boolean circuit of size at most s with negations only at the inputs. Let C be a skew Boolean circuit of size $s+1$ with negations only at the inputs, and g be the output gate of C with inputs from gates g_1 and g_2. We then have two cases. First suppose that g is an OR gate. Consider the two subcircuits C_1 and C_2 with g_1 and g_2 as their output, respectively. Since C_1 and C_2 are also skew circuits of sizes at most s with negations only at the inputs, we have $\deg(C_1) \leq s+1$ and $\deg(C_2) \leq s+1$ by induction. Then we have $\deg(C) = \max(\deg(C_1), \deg(C_2)) \leq s+1 \leq s+2$.

Now consider the case that g is an AND gate. We are done if both g_1 and g_2 are inputs of C. So assume that exactly one of g_1 and g_2 is an input of C. Without loss of generality, let g_1 be an input of C. Then we have $\deg(g_1) = 0$ or 1. Consider the subcircuit C_2 with g_2 as its output. Since C_2 is a skew circuit of size at most s, we have $\deg(C_2) \leq s+1$ by induction. Then we have $\deg(C) = \deg(g_1)+\deg(g_2) \leq s+2$. Thus the claim is true by induction. □

Theorem 6.2.2. *Let C be a skew Boolean circuit with negations only at the inputs such that C has size s. Then there exists a Boolean circuit of size $O(s^3)$ and depth $O((\log s)^2)$ computing the same function as C.*

Proof. Follows directly from Corollary 6.1.5 and Lemma 6.2.1 □

Note that Theorem 6.2.2 gives an alternative proof for $NL \subseteq SAC^1$ [11] in the non-uniform setting, since the construction in Corollary 6.1.5 can be immediately translated into semi-unbounded fanin circuits. The theorem also implies that every

skew Boolean circuit with negations only at the inputs can be parallelized well, since we only have a cubic increase in the size. Furthermore, Lemma 6.2.1 together with Theorem 2.5.5 by Allender, Jiao, Mahajan, and Vinay [4] give an alternative proof for $NL \subseteq SAC^1$ [11] in the uniform setting. Lastly note that the algorithms in Theorem 2.5.3 [137] and Theorem 2.5.5 [4] do not guarantee to construct a skew circuit. So we cannot simulate C by a skew circuit, as we did in the case of planar circuits, layered circuits, and synchronous circuits.

6.3 Multilinear Boolean Circuits

Raz and Yehudayoff [113] proved Theorem 2.5.7 that the construction by Valiant, Skyum, Berkowitz and Rackoff [137] for Theorem 2.5.3 preserves syntactic multilinearity for arithmetic circuits over a field, which implies that syntactic multilinear-P equals syntactic multilinear NC^2 for arithmetic circuits over a field. In this section, we will show that the result by Raz and Yehudayoff [113] also holds for Boolean circuits. One could possibly extend the technique by Raz and Yehudayoff [113] to semirings, but instead here we give a more direct proof.

A Boolean circuit is said to be *multilinear* if for any AND gate g in the circuit with inputs from g_1 and g_2, the two sets of input variables for the two subcircuits with g_1 and g_2 as their respective output gate are disjoint. To be more specific, let $var(g)$ denote the set of input variables that can reach gate g. Then for any AND gate $g \leftarrow g_1 \wedge g_2$ in a multilinear circuit, we must have $var(g_1) \cap var(g_2) = \emptyset$. Note that for Boolean circuits, multilinearity is defined the same as syntactic multilinearity instead of considering the degree of polynomials. Otherwise with the alternative definition, every Boolean circuit would be multilinear.

Multilinear Boolean circuits were studied in [125], [108], and [75]. (Here we use the definition from [125] and [75].) Multilinear arithmetic circuits were studied in [112] and [111]. Any Boolean function can be computed by a multilinear Boolean circuit (simply from the DNF representation of the function). The following lemma shows that the degree of a multilinear Boolean circuit is at most linear in its input length.

Lemma 6.3.1. *Let C be a multilinear Boolean circuit with n input variables such that negations of C are only at the inputs, and C has at least one gate. Then the degree of C is at most n.*

Proof. We shall prove by induction on C's size, s. Suppose that $s = 1$. Then the degree of C is either 1 or 2, so the claim is true in the base case. Now suppose that the claim is true for all multilinear Boolean circuit of size at most s with negations only at the inputs. Let C be a multilinear Boolean circuit of size $s+1$ with negations only at the inputs such that C has input length n. Let g be the output gate of C with inputs from g_1 and g_2. If g is an OR gate, then we have

$$\deg(g) = \max(\deg(g_1), \deg(g_2)) \leq n,$$

since the degrees of g_1 and g_2 are at most n from the induction hypothesis. So we are done in this case.

Now suppose that g is an AND gate. Since C is multilinear, we have $var(g_1) \cap var(g_2) = \emptyset$ and $|var(g_1)| + |var(g_2)| = n$. Then by induction hypothesis, we have

$$\deg(g) = \deg(g_1) + \deg(g_2) \leq |var(g_1)| + |var(g_2)| = n.$$

So the claim is also true in this case. □

We now show that given a multilinear Boolean circuit, the algorithm in [137] will produce a multilinear circuit. For completeness, we start with several definitions from [137].

Definition 6.3.2. [137] Let C be a Boolean circuit with negations only at the inputs, $V = \{v_i\}$ be the gates in C, $X = \{x_1, \ldots, x_n\}$ be the set of input variables, and $\overline{V} = V \cup X$. We then define the following.

1. $d(v_i)$ is the degree of v_i.
2. Without loss of generality, for each $w \leftarrow w' \circ w''$ in C, where $\circ \in \{\vee, \wedge\}$, we assume $d(w') \geq d(w'')$.
3. For $a > 0$, define $V_a = \{t \in V : \ d(t) > a, t \leftarrow t' \wedge t'', d(t') \leq a\}$.
4. For $a > 0$, define $V'_a = \{t \in V : \ d(t) > a, t \leftarrow t' \vee t'', d(t'') \leq a\}$.
5. $f(v_i)$ is the formal polynomial (over X) computed at gate v_i.
6. Let $v, w \in \overline{V}$, $f(v; w)$ is defined inductively on the depth of w.

(a) If $v = w$, then $f(v; w) = 1$.

(b) Otherwise if $v \neq w$ and $w \in F \cup X$, then $f(v; w) = 0$.

(c) Otherwise if $w \leftarrow w' \vee w''$, then $f(v; w) = f(v; w') \vee f(v; w'')$.

(d) Otherwise if $w \leftarrow w' \wedge w''$, then $f(v; w) = f(w'') \wedge f(v; w')$.

We need the following fact, which is easy to verify.

Fact 6.3.3. *Let C be a Boolean circuit with negations only at the inputs computing $f(v_{out})$, where v_{out} is the output gate of C. Then $d(v_i) \leq d(v_{out})$ for all gate v_i.*

We also need the following lemma for $f(v; w)$.

Lemma 6.3.4. *Given any $v, w \in \overline{V}$ such that there is no path from v to w in C, then we have $f(v; w) = 0$.*

Proof. We shall prove by induction on w. If $v = w$, then $f(v; w) = 1 \neq 0$. If $v \neq w$ and $w \in X$, then there is no path from v to w and $f(v; w) = 0$ by definition. So the claim is true in the two base cases.

Now suppose that we have $w \leftarrow w' \vee w''$ or $w \leftarrow w' \wedge w''$. Since there is no path from v to w, there is no path from v to w' or from v to w''. From induction hypothesis, we then have $f(v; w') = f(v; w'') = 0$, which implies that $f(v; w) = 0$ in both cases. Thus the claim is true by induction. □

The following lemma says that $f(v; w)$ does not depend on any variable from $var(v)$.

Lemma 6.3.5. *Given any $v, w \in \overline{V}$ with $d(w) < 2d(v)$, $f(v; w)$ depends only on input variables from $var(w) \setminus var(v)$.*

Proof. We prove this by induction on w. If $v = w$ (same node), then we have $f(v; w) = 1$ and $var(w) \setminus var(v) = \emptyset$. If $v \neq w$ (different nodes) and $w \in X$, then we have $f(v; w) = 0$ and $var(w) \setminus var(v) = \{w\}$ or $\emptyset$. So the claim is true in the two base cases.

Suppose that $w \leftarrow w' \vee w''$. Then we have $f(v; w) = f(v; w') \vee f(v; w'')$. From induction hypothesis, $f(v; w')$ and $f(v; w'')$ only depends on variables from $var(w') \setminus var(v)$ and $var(w'') \setminus var(v)$, respectively. Thus $f(v; w)$ only depends on variables in $var(w) \setminus var(v)$, since we have $var(w) = var(w') \cup var(w'')$.

Now suppose that $w \leftarrow w' \wedge w''$. Then we have $f(v; w) = f(w'') \wedge f(v; w')$. There are three cases.

1. There is no path from v to w. Then by Lemma 6.3.4, we have $f(v;w) = 0$. So the claim is true in this case.

2. There is a path from v to w'. Then we have $var(v) \subseteq var(w')$, which implies that $var(v) \cap var(w'') = \emptyset$ by the definition of a multilinear circuit. By induction hypothesis, $f(v;w')$ only depends on variables from $var(w') \setminus var(v)$. So $f(v;w)$ only depends on variables from

$$\begin{aligned} & var(w'') \cup (var(w') \setminus var(v)) \\ = \; & var(w'') \cup (var(w') \cap \overline{var(v)}) \\ = \; & (var(w'') \cup var(w')) \cap (var(w'') \cup \overline{var(v)}) \\ = \; & var(w) \cap \overline{var(v)} \\ = \; & var(w) \setminus var(v). \end{aligned}$$

   So the claim is also true in this case.

3. There is no path from v to w', but there is a path from v to w''. So we have $d(v) \leq d(w'')$. But then $d(w) = d(w') + d(w'')$ and $d(w'') \leq d(w')$ imply that $d(w) \geq 2d(v)$. So this case will never occur since we assumed $d(w) < 2d(v)$.

Since we have proved the claim in all possible cases, it is true by induction. □

Here we prove the multilinear version of the main lemma from [137].

Lemma 6.3.6. *Let C be a multilinear Boolean circuit with negations only at the inputs, V be the gates of C, X be the set of input variables, $a > 0$, $v, w \in \overline{V} = V \cup X$, and $d(v) \leq a < d(w)$. Then we have*

$$f(v;w) = \bigvee_{t \in V_a} (f(v;t) \wedge f(t;w)) \vee \bigvee_{t \in V'_a} (f(v;t'') \wedge f(t;w))$$

and

$$f(w) = \bigvee_{t \in V_a} (f(t) \wedge f(t;w)) \vee \bigvee_{t \in V'_a} (f(t'') \wedge f(t;w))$$

Furthermore, in each conjunction, the two functions take inputs from disjoint subsets of X provided that $d(w) < 2d(v)$.

Proof. The validity of the two equalities was established in [137], so we only need to show that the two functions in each conjunction takes inputs from disjoint subsets of the input variables. By Lemma 6.3.5, we have

1. $f(t;w)$ only depends on variables in $var(w) \setminus var(t)$ (which does not intersect $var(t'')$);
2. $f(v;t)$ only depends on variables in $var(t) \setminus var(v)$;
3. $f(v;t'')$ only depends on variables in $var(t'') \setminus var(v) \subseteq var(t) \setminus var(v)$.

Then the input variables $f(v;t)$, $f(v;t'')$, $f(t)$, $f(t'')$ depend on and the input variables $f(t;w)$, $f(t;w)$, $f(t;w)$, $f(t;w)$ depend on are disjoint, respectively. □

Theorem 6.3.7. *Let C be a multilinear Boolean circuit with negations only at the inputs such that C has size s. Then there exists a multilinear Boolean circuit C' with negations only at the inputs of size $O(s^3)$ and depth $O(\log^2 s)$ computing the same function as C.*

Proof. We first sketch the construction given in [137] of the new Boolean circuit. At stage 0 we compute all $f(w)$ and $f(v;w)$ with degree at most 1. At stage $i+1$ we compute all $f(w)$ and $f(v;w)$ with degree in $(2^i, 2^{i+1}]$ by Lemma 6.3.6:

$$f(v;w) = \bigvee_{t \in V_a} (f(t'') \wedge f(v;t') \wedge f(t;w)) \vee \bigvee_{t \in V'_a} (f(v;t'') \wedge f(t;w))$$

(with $a = 2^i$) and

$$f(w) = \bigvee_{t \in V_a} (f(t') \wedge f(t'') \wedge f(t;w)) \vee \bigvee_{t \in V'_a} (f(t'') \wedge f(t;w))$$

(with $a = 2^i + d(v)$). By the definition of V_a and V'_a, each $f(t')$, $f(t'')$, $f(t;w)$, $f(v;t')$ and $f(v;t'')$ has already been computed. So in each stage we increase the depth by $O(\log s)$. By Fact 6.3.3, there are at most $O(\log d)$ stages. Therefore the depth of the new circuit is $O(\log d \log s)$. The analysis is the same as given in [137].

Observe that the above construction preserves multilinearity, since all functions in each conjunction depend on disjoint sets of input variables. Now let n be the number of input variables of C. By Lemma 6.3.1, the degree of C is at

most n. Then the new multilinear Boolean circuit C' has size $O(s^3)$ and depth $O(\log s \log n) = O(\log^2 s)$ that computes the same function as C. Furthermore, all negations of C' are only at the inputs. □

Corollary 6.3.8. *For Boolean circuits, we have* $Multilinear - P = Multilinear - NC^2$*, where* $Multilinear - P$ *is the class of languages decided by multilinear circuit families of polynomial size, and* $Multilinear - NC^2$ *is the class of languages decided by multilinear circuit families of polynomial size and* $O(\log^2 n)$ *depth.*

Chapter 7

Space-Efficient Algorithms

SUBSET SUM, decision version Given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers and $t \in Z^+$, decide if there exists a subset $S' \subseteq S$ such that the sum of all numbers in S' equals t. We denote by n the length of the input represented in binary, that is $n = \sum_{i=1}^{m} \log a_i + \log t$.

One can also consider the **search version**, where if such subset exists, we also have to output an appropriate subset. Another variant is an optimization version, defined as follows.

SUBSET SUM, optimization version [28] Given a finite set S of positive integers and $t \in Z^+$, output a subset of S whose sum is as large as possible but no larger than t. Output NO if no such subset can be found.

The classical exact solution for the decision, search and optimization versions of the SUBSET SUM problem is to use dynamic programming (Dantzig [29], Bellman [9]), which runs in time $O(mt)$ and space $O(t)$ in the unit-cost RAM model, and can be made to run in time $O(mt \log t(\log m + \log\log t))$ and space $O(t + \log m)$ on the Turing machine model. This is *pseudo*-polynomial in m, that is, if the value of t is bounded by a polynomial in m, then it runs in polynomial time and space. Otherwise the algorithm runs in super-polynomial time and space. See Garey and Johnson ([47] Section 4.2) for details.

Approximation algorithms for the SUBSET SUM problem have also been extensively studied. Let t^* be the value of the sum in an optimal solution to the SUBSET SUM problem (optimization version), and let $\hat{t}$ be the value of the sum in

the solution computed by the approximation algorithm. Note that t^* is not known in advance. Define the *approximation factor* to be $\max(\frac{t^*}{\hat{t}}, \frac{\hat{t}}{t^*})$. An *approximation scheme* is an approximation algorithm whose inputs are the problem instance and a value ϵ, such that for any fixed $\epsilon > 0$, the algorithm, achieves $1 + \epsilon$ approximation factor. We say that an approximation scheme is a *polynomial-time approximation scheme* if for any $\epsilon > 0$, the running time is polynomial in n, where n is the input length. An approximation scheme is *FPTAS (Fully Polynomial-Time Approximation Scheme)* if the running time is polynomial in both $1/\epsilon$ and n. See Vazirani [138] and Cormen et.al [28] for detailed discussions.

Ibarra and Kim [62] gave an FPTAS for the problem with running time $O(m/\epsilon^2)$ and space $O(m + 1/\epsilon^3)$. Gens and Levner [50] gave an alternative FPTAS algorithm running in time $O(m/\epsilon + 1/\epsilon^3)$ and space $O(m + 1/\epsilon)$. They also gave an FPTAS algorithm for the MINIMIZATION SUBSET SUM problem in the same time and space bound. The MINIMIZATION SUBSET SUM is similar to the SUBSET SUM optimization problem, but we want a subset of S whose sum is as small as possible but no smaller than t. Both results by Ibarra and Kim, and Gens and Levner use dynamic programming. Also see the survey by Gens and Levner [49]. As far as we know, the current best FPTAS algorithm for the SUBSET SUM problem is given by Kellerer, Mansini, Pferschy, and Speranza [71], which runs in time $O(\min(m/\epsilon, m + 1/\epsilon^2 \log \frac{1}{\epsilon}))$ and space $O(m + 1/\epsilon)$. See Woeginger and Yu [150] and Bazgan, Santha, and Tuza [7] for further variants and FPTAS for the SUBSET SUM problem. All these results are obtained under the unit-cost RAM model.

We also consider approximation algorithms for the decision and search versions of the SUBSET SUM problem.

γ-approximate SUBSET SUM, decision/search version Given a finite set S of positive integers, $t \in Z^+$, and $0 < \gamma < 1$, decide if there exists a subset $S' \subseteq S$ such that

$$(1 - \gamma)t \leq \sum_{a_i \in S'} a_i \leq (1 + \gamma)t.$$

For the search version, output such a subset if it exists.

One special case of the SUBSET SUM problem is the PARTITION problem, which was one of Karp's first 21 NP-complete problems [69]. The problem is defined

as follows.

PARTITION, decision/search version: Given a finite set S of positive integers, decide if there is a subset $S' \subseteq S$ such that

$$\sum_{a_i \in S'} a_i = \sum_{b_i \in S \setminus S'} b_i.$$

For the search version, output such a subset if it exists.

In the SUBSET SUM problem, if we set $t = \frac{A}{2}$, where $A = \sum_{a_i \in S} a_i$, then we get the PARTITION problem.

As for the SUBSET SUM problem, the classical solution for PARTITION is to use dynamic programming (Dantzig [29], Bellman [9]), which runs in time $O(mt)$ and space $O(t)$.

The classical algorithms [29, 9] solving the decision version of the SUBSET SUM problem run in pseudo-polynomial time *and* space. Lokshtanov and Nederlof [85] gave an algorithm that runs in pseudo-polynomial time and *polynomial* space. More precisely, let $\tilde{O}(t(n))$ denote $O(t(n) \log^k t(n))$ for some constant $k > 0$. Then their algorithm for the SUBSET SUM problem runs in time $\tilde{O}(m^3 t \log t)$ and space $\tilde{O}(m^2)$ on the unit-cost RAM model. They also gave an algorithm for the unbounded Knapsack problem (see next section for definitions), which runs in time $\tilde{O}(m^4 ct \log^2(ct))$ and space $\tilde{O}(m^2 \log(ct))$ on the unit-cost RAM model, where c, t are the capacity and target value, respectively.

As mentioned above, the SUBSET SUM problem can be solved in polynomial time if all the numbers are bounded by a polynomial in n. This means that if the numbers are represented in unary format, then we can always solve the problem in polynomial time. This interesting variant is called the UNARY SUBSET SUM, where all the input numbers are given in unary representation. Elberfeld, Jakoby, and Tantau [35], and Kane [66] showed that UNARY SUBSET SUM is in L. Later Elberfeld, Jakoby, and Tantau [36] showed that the problem is in NC^1 and is actually TC^0 complete. On the Turing machine model, when the input is represented in binary form, both the algorithms given by Elberfeld, Jakoby, and Tantau [35] and Kane [66] yield an algorithm running in time polynomial in n and t and in space $O(\log n + \log t)$.

Limaye, Mahajan, and Sreenivasaiah [81] considered approximating the UNARY

SUBSET SUM problem as follows. Given γ, B, and S such that each $a_i \in S$ satisfies $1 \leq a_i \leq B$, output YES if and only if there exists a subset $S' \subseteq S$ such that

$$B(1-\gamma) \leq \sum_{a_i \in S'} a_i \leq B(1+\gamma).$$

They gave a 1-pass streaming algorithm for this variant of the UNARY SUBSET SUM problem that uses space $O(\frac{\log B}{\gamma})$.

While in the definition of SUBSET SUM (and approximate SUBSET SUM) in general we do not require that all numbers are at most t, the assumption $a_i \leq B = t$ in [81] does not limit generality in this respect: we can simply ignore elements $a_i > (1+\gamma)t$ in the stream since they cannot participate in a sum with value $\leq (1+\gamma)t$. Thus, Limaye, Mahajan, and Sreenivasaiah [81] give a 1-pass streaming algorithm for γ-approximate SUBSET SUM that runs in $O(\log n)$ space for any constant γ as long as t is polynomial in n.

7.1 Our Results

The above mentioned space efficient algorithms for SUBSET SUM putting UNARY SUBSET SUM in logspace [35, 66] and the space efficient approximation algorithm of Limaye, Mahajan, and Sreenivasaiah [81] only consider the decision version of the problem, they do not find a subset if it exists.

Note that an algorithm solving the decision version of SUBSET SUM can be used to also output an appropriate subset as follows: Fix an arbitrary element a, and run the algorithm for the decision version (with $t - a$ instead of t) to check if there is a subset with the required sum that contains a. Repeat this process until an element is found that can participate in the subset, and then repeat again on a smaller collection of elements. This procedure eventually finds an appropriate subset, and it runs the algorithm for the decision version no more than n^2 times. However, the input to the decision version needs to be changed for the various runs of the algorithm, and the algorithm needs to keep track of the elements already included in the subset by keeping them in working memory. Thus, the procedure described above will use linear space.

We observe that with a more careful reduction, any space efficient algorithm solving the decision version of SUBSET SUM can be used to obtain a space efficient

algorithm for solving the search and optimization versions of SUBSET SUM. The algorithms of [35, 66] yield algorithms that solve the search and optimization versions of SUBSET SUM exactly, in time polynomial in n and t, and space $O(\log n + \log t)$. This improves the space bound of the classical exact solutions from space $O(t)$ to $O(\log n + \log t)$. Note that this also gives an improvement over the $\tilde{O}(m^2)$ space bound by Lokshtanov and Nederlof [85]. The running time is pseudopolynomial in n: it is polynomial if the value t is polynomial in n.

For approximating SUBSET SUM, we can improve the running time, depending on the approximation factor, but independent of the value of t. We give space efficient *fully polynomial time* approximation algorithms for both the search and optimization versions of SUBSET SUM.

We also observe a connection between approximation algorithms for PARTITION and finding balanced graph separators. Based on this connection and the space efficient approximation algorithms for SUBSET SUM, we give space efficient algorithms to construct balanced graph separators.

We also consider variants of the knapsack problem. Similarly to SUBSET SUM one can consider decision, search, and optimization versions. The *decision version* of the 0-1 knapsack problem can be defined as follows. Given a capacity $c \in Z^+$, a target value $t \in Z^+$, and a set of size m, where each element has value a_i and weight w_i, output YES iff there is a subset $S \subseteq [1, m]$ such that $\sum_{i \in S} a_i = t$ and $\sum_{i \in S} w_i \leq c$. See Karp [69]. In the search version, we have to output the subset S if it exists. The *optimization version* is defined as follows. Given a capacity $c \in Z^+$, and a set of size m, where each element has value a_i and weight w_i, find a subset $S \subseteq [1, m]$ such that $\sum_{i \in S} w_i \leq c$ and $\sum_{i \in S} a_i$ is as large as possible. See Section 6.1 of Garey and Johnson [47]. Note that the optimal value is not known in advance. An equivalent definition is to maximize $\sum_{i=1}^{m} a_i x_i$ with the constraints $\sum_{i=1}^{m} w_i x_i \leq c$ and $x_i \in \{0, 1\}$ for $1 \leq i \leq m$. Note that we get the SUBSET SUM problem as a special case of 0-1 knapsack by letting $w_i = a_i$ and $c = t$. One common generalization of the 0-1 knapsack problem is the *integer knapsack problem*, where every x_i is a non-negative integer. This is also known as the *unbounded knapsack problem*, since the number of times we can pick each item is unbounded.

Another interesting variant is the *change-making problem*. The decision version is defined as follows. Note that this is a special case of the unbounded knapsack problem by letting $w_i = 1$. Given positive integers a_i for $1 \leq i \leq m$, t, and c, output

YES iff $\sum_{i=1}^{m} a_i x_i = t$ and $\sum_{i=1}^{m} x_i \leq c$, where $x_i \geq 0$ is an integer. This corresponds to paying a fixed amount t using fewest coins as possible, where a_i are the coin denominations. The change-making problem was first studied by Lueker [86], who showed that the problem is NP-hard. It was later studied by Kozen and Zaks [74], and Adamaszek and Adamaszek [1]. Also see Martello and Toth [91], and Kellerer and Pferschy and Pisinger [72] for a comprehensive coverage of the problem.

7.2 Approximation Algorithms for SUBSET SUM

Our approximation algorithms for the search and optimization versions of SUBSET SUM are based on repeatedly running the algorithm of Limaye, Mahajan, and Sreenivasaiah [81], but unlike in the simple reduction to the decision version described above, we do not keep in working memory the elements found so far, and each time we run the algorithm on the same set S.

Theorem 7.2.1. *There is a deterministic algorithm, that given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \gamma < 1$, outputs a subset $S' \subseteq S$ such that*

$$(1-\gamma)t \leq \sum_{a_i \in S'} a_i \leq (1+\gamma)t.$$

if such subset exists, otherwise it outputs "NO". The algorithm runs in $O(\frac{1}{\gamma}mn) = O(\frac{1}{\gamma}n^2)$ time, and $O(\frac{1}{\gamma}\log t + \log n)$ space, where $n = \sum_{i=1}^{m} \log a_i + \log t$.

Note that the algorithm runs in polynomial time and logarithmic space for any constant $\gamma > 0$ as long as the value of t is polynomial in n. For this, we do not need to require that all the values a_i are polynomial in n, since we can just ignore any value that is larger than $(1+\gamma)t$, and we don't have to use workspace for keeping any of the a_i-s with larger values.

Proof. Our proof is based on the following observations about properties of the algorithm of Limaye, Mahajan, and Sreenivasaiah [81].

Let $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \gamma < 1$. Suppose that the algorithm in [81] answers "YES" on this input, and let a_r be the last element read from the stream before the algorithm stops. Recall that the task

of the algorithm in [81] is to determine whether a subset S' exists such that

$$(1-\gamma)t \leq \sum_{a_i \in S'} a_i \leq (1+\gamma)t.$$

We refer to such set S' as a *good* set, with respect to t and γ.

Observation 7.2.2. *There exists a good set with respect to t and γ that contains a_r.*

Observation 7.2.3. *If $(1-\gamma)t - a_r > 0$, then there is a subset H' of $H = \{a_1, \ldots, a_{r-1}\}$ such that the sum of the elements in H' satisfies*

$$(1-\gamma)t - a_r \leq \sum_{a_i \in H'} a_i \leq (1+\gamma)t - a_r.$$

Observation 7.2.4. *Let $P = \{a_1, \ldots, a_r\} \subseteq S = \{a_1, \ldots, a_m\}$ be the first r elements of S read when reading the set S as a stream. If P contains a good subset (with respect to a given t and γ) then the algorithm stops with YES (when running it on S with t and γ), and the last element read is a member of P.*

These observations can be proved by induction on r, similarly to the proof of correctness of the algorithm in [81].

Our algorithm will repeatedly run the algorithm of Limaye, Mahajan, and Sreenivasaiah [81]. To keep the workspace used by the algorithm small, it is important that we always run the algorithm on the same set S, we only change the values of t and γ for the various runs.

First, we run the algorithm on S, γ, and t. If the algorithm answers "NO", we stop and output "NO".

Suppose that the algorithm answers "YES" on this input, and let a_r be the last element read from the stream before the algorithm stops. If $r = 1$, we write a_1 on the output tape and stop.

If $r > 1$, we write a_r on the output tape. By Observation 7.2.2, we know that a_r can be part of a correct output, so we can write it on the output tape, even though we may not have finished finding a good set yet.

If $(1-\gamma)t \leq a_r \leq (1+\gamma)t$, then we stop.

Otherwise, note that $(1-\gamma)t - a_r > 0$ must hold. Next, we run the algorithm again on S, and $t' = t - a_r$ with $\gamma' = \gamma \frac{t}{t-a_r}$. Note that we chose t' and γ' so that the algorithm in this run checks if there is a subset of S such that the sum of its elements is at least $(1-\gamma)t - a_r$ and at most $(1+\gamma)t - a_r$. By Observation 7.2.3, we know that the algorithm must terminate with "YES".

Let $a_{r'}$ be the last element read. By Observations 7.2.3 and 7.2.4, we know that $r' < r$.

We iterate the above process, until $r' = 1$. Since $r' < r$ in each iteration, the process must terminate in at most n iterations. When the procedure terminates, a good set is listed on the output tape. □

Theorem 7.2.5. *There is a deterministic approximation scheme for SUBSET SUM optimization, that for any $\epsilon > 0$ achieves approximation factor $1+\epsilon$, and runs in time $O(\frac{1}{\epsilon}n^2)$ and space $O(\frac{1}{\epsilon}\log t + \log n)$.*

Proof. First we do a "binary search" to find the value of the optimal sum t^* approximately, by repeatedly running the algorithm of Limaye, Mahajan, and Sreenivasaiah [81] that solves the approximate decision version.

Given an instance of SUBSET SUM optimization with S, $t \in Z^+$, and $\epsilon > 0$, we run the algorithm of [81] on S and pairs t_i, γ_i for $i = 1, \ldots, \ell$, defined as follows.

We use the notation $t_0 = \frac{t}{2}$. Let $t_1 = \frac{3}{4}t$, $\gamma_1 = \frac{1}{4}$, that is, in the first run, we check if there exists a subset with sum between $\frac{t}{2}$ and t. Next we let $\gamma_{i+1} = \frac{1}{2}\gamma_i$, and depending on the answer to the i-th run, we set $t_{i+1} = t_{i-1} - \gamma_i t$ if the answer is NO, or $t_{i+1} = t_i + \gamma_{i+1}t$, if the answer is YES. (For example, if in the first run the answer is NO then $t_2 = \frac{t}{4}$, and if in the first run the answer is YES then $t_2 = \frac{3}{4}t + \frac{1}{8}t$.) We stop after running the algorithm with $\gamma_\ell \leq \frac{\epsilon}{2+\epsilon}$ for the first time, that is after $\ell \leq \log \frac{2+\epsilon}{\epsilon}$ iterations.

Let $t' = t_{\ell-1} - \gamma_\ell t$ if the answer is NO, or $t' = t_\ell$, if the answer is YES. Let $\gamma = \gamma_\ell$ and let $\delta = \frac{2\gamma}{1+\gamma}$.

Notice that the way our "binary search" proceeds we have

$$(1-\gamma)t' \leq t^* \leq (1+\gamma)t'.$$

where t^* denotes the value of the optimal sum.

At this point, we know that there exists a good subset S' of S with respect

to t' and γ, that is

$$(1-\gamma)t' \leq \sum_{a_i \in S'} a_i \leq (1+\gamma)t'.$$

Next we run our algorithm from Theorem 7.2.1 that outputs such subset. Note that we have chosen the parameters so that $(1-\delta)t^* \leq (1-\gamma)t'$. By definition, we cannot find a sum larger than the optimal sum. Thus we obtain an approximation scheme with approximation factor $\frac{1}{1-\delta} = 1+\epsilon$.

Note that we can assume $\epsilon \geq \frac{1}{t}$ (since using $\epsilon = \frac{1}{t}$ gives an exact solution). Thus $\ell = O(\log t) = O(n)$, and the total running time of our algorithm is $O(\frac{1}{\epsilon}n^2)$. □

In the special case of $\gamma = \frac{1}{3}$, we get an even simpler algorithm than what would follow by substituting $\gamma = \frac{1}{3}$ in Theorem 7.2.1. This algorithm is comparable in performance to the streaming algorithm of Limaye, Mahajan, and Sreenivasaiah [81], but while [81] only decides existence of an appropriate subset, our algorithm also finds the set if it exists.

Theorem 7.2.6. *There is a deterministic 2-pass streaming algorithm, that given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers and $t \in Z^+$, outputs a subset $S' \subseteq S$ such that*

$$\frac{2}{3}t \leq \sum_{a_i \in S'} a_i \leq \frac{4}{3}t.$$

if such subset exists, otherwise it outputs "NO". The algorithm runs in linear time, and $O(\log t + \log n)$ space, where $n = \sum_{i=1}^{m} \log a_i + \log t$.

Proof. This is a special case of Theorem 7.2.1 with $\gamma = \frac{1}{3}$. However, in this special case we get an even simpler algorithm, which is a 2-pass streaming algorithm.

During the algorithm we will only keep in working memory the value of a current sum s and some counters.

During the first pass, we start by initializing s to 0. For each $i = 1, \ldots, n$, we look at the value a_i on the input tape. There are 3 cases:

1. If $a_i > \frac{4}{3}t$ we move to the next element.
2. If $\frac{2}{3}t \leq a_i \leq \frac{4}{3}t$ we output a_i and stop.
3. If $a_i < \frac{2}{3}t$, we let $s = s + a_i$ Then, if $s < \frac{2}{3}t$ we move to the next element. If we run out of elements and still have $s < \frac{2}{3}t$, we stop and output NO. If $\frac{2}{3}t \leq s \leq \frac{4}{3}t$

we move to the second pass, see below. Note that $s > \frac{4}{3}t$ is not possible, since the previous sum was less than $\frac{2}{3}t$ and we added a value less than $\frac{2}{3}t$.

The only reason we need a second pass is because during the first pass we did not know yet if we have to answer NO or output a set, and the output tape is write only.

In the second pass we proceed exactly as in the first pass, but in addition, every time we add an element a_i to the sum, we also write it on the output tape. The algorithm stops when we reach for the first time a value $s \geq \frac{2}{3}t$. At this point, we also have the members of a good subset listed on the output tape.

We argue that the algorithm finds a good subset if one exists as follows. Elements larger than $\frac{4}{3}t$ cannot participate in a good subset, and if a single element a_i is in the correct range we find it during the first scan. If no such element exists, than all members of a good subset must be less than $\frac{2}{3}t$. If the first scan terminates with s still less than $\frac{2}{3}t$ that means that the sum of all the elements less than $\frac{2}{3}t$ is a value also less than $\frac{2}{3}t$. But then the sum of any subset of these elements is also less than $\frac{2}{3}t$, and no good subset exists.

This concludes the proof of the theorem. □

We can extend this argument to other values of γ and keep the running time linear in n for small values of $\frac{1}{\gamma}$. In particular, we still get a 2-pass algorithm for $\gamma = \frac{1}{5}$ with $O(\log t + \log n)$ space. But we cannot keep the algorithm linear time and logarithmic space for arbitrary constant γ, and as $\frac{1}{\gamma}$ gets larger, our quadratic time algorithm of Theorem 7.2.1 gives better running time, while still using logarithmic space for arbitrary constant γ.

7.3 Finding Balanced Separators

Balanced separators are used in VLSI circuit design (Ullman [136]), algorithms on planar graphs (Lipton and Tarjan [84, 83], Miller [97]), graphs with excluded minors (Alon, Seymour, and Thomas [5]), and in general in algorithmic applications on graphs that involve a divide and conquer strategy.

Informally, a *balanced* separator of a graph G is a set of nodes whose removal yields two disjoint subgraphs of G that are comparable in size. More formally, balanced separators are defined as follows.

Definition 7.3.1. (Lipton and Tarjan [84]) Let $G = (V, E)$ be an undirected graph. A set of nodes $S \subseteq V$ is a *balanced* separator of G if the removal of S disconnects G into two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, that satisfy $|V_i| \geq \frac{1}{3}|V|$ for $i = 1, 2$.

We say that S is a balanced h-separator of G if S is a balanced separator of G and S consists of h vertices. We say that G is h-separable, if it has a balanced h-separator.

Note that the definition can be easily extended to directed graphs. Also, instead of requiring that the size of both components is between $\frac{1}{3}|V|$ and $\frac{2}{3}|V|$, we can consider components with size closer to $\frac{1}{2}|V|$.

Definition 7.3.2. Let $G = (V, E)$ be an undirected graph. A set of nodes $S \subseteq V$ is an γ- *balanced* separator of G if the removal of S disconnects G into two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, that satisfy $|V_i| \geq (1 - \gamma)\frac{|V|}{2}$ for $i = 1, 2$.

The two subgraphs G_1 and G_2 could be disconnected within themselves. This implies that any graph $G = (V, E)$ has a γ-balanced separator of size $|V|$, for any constant $1 > \gamma > 0$.

We observe that space efficient approximation algorithms for the decision version of the PARTITION problem yield space efficient algorithms to find balanced separators. Moreover, we find a separator of the smallest size, without knowing the size of the smallest separator in advance.

Theorem 7.3.3. *There is a deterministic algorithm, that given an undirected graph $G = (V, E)$ and $0 < \gamma < 1$ as input, if G has γ-balanced separators of size h, then the algorithm outputs a γ-balanced separator of size at most h. The algorithm runs in time $O(\frac{1}{\gamma}|V|^{(h+O(1))})$, and uses space $O((h + \frac{1}{\gamma}) \log |V|)$.*

Note that the value h is not provided as input to the algorithm, and the algorithm outputs a separator of the smallest possible size. The algorithm runs in polynomial time and logarithmic space for graphs with constant size separators.

As far as we know, this is the first space efficient algorithm that finds balanced separators in the sense of Definition 7.3.1. In [35], Elberfeld, Jakoby, and Tantau considered a different notion of separators, (also commonly referred to as balanced separators in the literature) that we call *component-wise* balanced separators, to distinguish them from the definition of Lipton and Tarjan [84] that we use.

Definition 7.3.4. [35] Let $G = (V, E)$ be an undirected graph. A set of nodes $S \subseteq V$ is a *component-wise* balanced separator of G if the removal of S disconnects G into subgraphs $G_i = (V_i, E_i)$ that satisfy $|V_i| \leq \frac{1}{2}|V|$ for each i.

Elberfeld, Jakoby, and Tantau [35] observed the following simple space efficient algorithm for finding *component-wise* balanced separators. Enumerate all possible subsets S of V first by size and then by lexicographic order for each size, starting from subsets of size 1. For each subset S use Reingold's algorithm [115] that solves undirected connectivity in logspace to compute the size of each component in the graph obtained by removing S from G. Output the first subset such that the size condition is satisfied by each component. It is clear that this algorithm can compute the component-wise balanced separator with minimum size, and uses $O(h \log |V|)$ space if the smallest component-wise separator has size h.

In particular, this algorithms runs in logspace for graphs that have constant size component-wise separators. This was used in [35] to find the tree decomposition of graphs with constant tree-width in logarithmic space.

However, for finding balanced separators in the sense of Definition 7.3.1, we also need to check whether the components obtained after removing a given subset of vertices can be grouped into two subsets of vertices, such that each has size at least $(1 - \gamma)\frac{|V|}{2}$. We observe that this corresponds to solving an instance of the γ-approximate PARTITION problem (a special case of γ-approximate SUBSET SUM), where the set $S = \{a_i\}$ consists of the sizes of the components $a_i = |V_i|$.

Moreover, since the values a_i are sizes of subsets of vertices of the graph, we are guaranteed that the value $A = \sum_i a_i$ is polynomial in $n = |V|$. Thus, our space efficient algorithms for approximating the SUBSET SUM and PARTITION problems yield space efficient algorithms for finding balanced separators. For graphs with constant size balanced separators our algorithm finds a minimum size separator in polynomial time and logarithmic space.

Since for finding balanced separators we only need to check if a subset of components with the appropriate sum of sizes exists, the linear time algorithm of Limaye, Mahajan, and Sreenivasaiah [81] solving the decision version of approximate SUBSET SUM can be used.

Putting all this together, we obtain the following proof of Theorem 7.3.3.

Proof. We define three logspace bounded Turing machines, to perform the following

tasks. Note that each machine uses the original graph G as input together with some additional inputs, including the list of vertices in a fixed set W. Other than the space used for storing their respective inputs, each machine uses at most $O(\log |V|)$ workspace, where V is the set of vertices of V. Moreover, the length of the output of each machine is $O(\log |V|)$.

For an undirected graph $G = (V, E)$, a set $W \subseteq V$ and a node $v \in V \setminus W$ we denote by G_W the graph obtained from G by removing the vertices in W and all edges adjacent to vertices in W. We denote by $G_W(v)$ the connected component of G_W that contains v.

The first Turing machine M_1 on input G, W and v computes the number of nodes connected to v in G_W, that is, M_1 outputs $|G_W(v)|$. This can be done by using Reingold's logspace algorithm for undirected connectivity [115] to check, one-by-one, for each node $u \in V \setminus W \setminus \{v\}$ if u is connected to v in G_W. We let Reingold's algorithm run on the original graph G (we do not store the representation of G_W), we simply ignore edges of G adjacent to vertices in W when running the algorithm. We reuse space for each repetition.

M_2 on input G and W outputs the number of connected components of G_W. Note that there is always at least one component, so we start by initializing a counter to $count = 1$. We assume that the first vertex belongs to the first component. Next, for $i = 2, \ldots, |V \setminus W|$, we check (using Reingold's algorithm) if $v_i \in V \setminus W$ is connected to any of the vertices $v_j \in V \setminus W$ for $1 \leq j < i$. If v_i is not connected to any of the previously considered vertices, we increase the counter, and move on to considering the next vertex. If v_i is connected to some previously considered vertex, then we move on to the next vertex without increasing the value of *count*. When all vertices in $V \setminus W$ have been considered, the value of *count* equals to the number of connected components of G_W.

Let G_W^ℓ denote the ℓ-th connected component of G_W, that is $G_W^1 = G_W(v_1)$, and $G_W^\ell = G_W(v_i)$, where i is the smallest integer such that v_i is not contained in any of the components $G_W^1, \ldots, G_W^{\ell-1}$.

M_3 on input G, W and ℓ computes the size of the ℓ-th connected component $|G_W^\ell|$, as follows. Initialize a counter to $count = 1$. Next, similarly to M_2, for $i = 2, \ldots, |V \setminus W|$, we check (using Reingold's algorithm) if $v_i \in V \setminus W$ is connected to any of the vertices $v_j \in V \setminus W$ for $1 \leq j < i$. If v_i is connected to some previously considered vertex, then we move on to the next vertex without increasing the value

of *count*. If v_i is not connected to any of the previously considered vertices, we increase the counter by 1, and then if $count < \ell$, move on to considering the next vertex. If $count = \ell$, we let $v = v_i$, and use M_1 on input G, W and v to compute $|G_W^\ell| = |G_W(v)|$.

We now define the algorithm in Theorem 7.3.3 as follows. We consider integers $1 \leq k \leq |V|$ in increasing order, and for each k we consider subsets $W \subseteq V$ of size k in lexicographic order. For fixed k and W, we use $k \log |V|$ workspace to keep the list of vertices in W. We use M_2 on G and W to compute the number of connected components of G_W. We denote this number by m. For $\ell = 1, \ldots, m$, let a_ℓ denote $|G_W^\ell|$, the size of the ℓ-th connected component.

We now need to check whether the connected components can be grouped into two subsets of vertices, such that each has size at least $(1-\gamma)\frac{|V|}{2}$. That is, we need to solve an instance of γ-approximate SUBSET SUM on input $S = \{a_\ell\}$ and $t = \frac{|V|}{2}$. However, we do not have enough work space to explicitly keep the list of integers in S.

We execute the algorithm to solve γ-approximate SUBSET SUM on $S = \{a_\ell\}$ and $t = \frac{|V|}{2}$, by using M_3 to compute the value a_ℓ when the algorithm needs it. Note that we only need to solve the decision version, so we can use the linear time algorithm of Limaye, Mahajan, and Sreenivasaiah [81]. Note that since the values a_i are sizes of subsets of vertices of the graph, we are guaranteed that the value $A = \sum_i a_i$ is polynomial in $|V|$. Thus, the space used for solving approximate SUBSET SUM is always bounded by $O(\log |V|)$. Storing the sets W takes $O(h \log |V|)$ space, where h is the size of the smallest γ-balanced separator. □

Observe that if G has γ-balanced separators of size h, we are guaranteed to find a γ-balanced separator of size at most h, and our algorithm does not need to know the value of h in advance.

Note that our running time is mostly dominated by the time we spend enumerating the $\binom{|V|}{k}$ possible subsets of size $k \leq h$. All other computations take polynomial time. For graphs with constant size balanced separators our algorithm finds a minimum size separator in polynomial time and logarithmic space.

Even the simpler problem of finding minimum size *component-wise* balanced separators is NP-hard for general graphs [61], and as far as we know, our running time is comparable to the running time of previously known algorithms to find

minimum size balanced separators. However, our algorithm is more space efficient than previous solutions.

Chapter 8

Relationships among Computational Measures

In this chapter, we study relationships between various computational measures. First we define several measures on Boolean circuits, and show how they are related to each other. Then we describe the construction of oblivious Turing machines by Pippenger and Fischer [103]. However, the circuit family given by Pippenger and Fischer was for the purpose of simulating Turing machines by circuits of small size, and the construction is not suitable to play the pebble game efficiently. So we modify the construction for the purpose of efficient pebbling. We present a pebbling strategy on these circuits, which reveal various relationships about simulating time and space of input-oblivious Turing machines. In addition, our construction also yields quadratic lower bounds on the simultaneous time and space product of input-oblivious Turing machines.

8.1 Measures on Boolean Circuits

Recall that in Section 1.1.2 and Section 3.1, we defined the depth and height of a node, as well as levels and layers of a circuit. We also defined synchronous circuits and layered circuits. In the following we define the *width* and *breadth* of general circuits. We note that the breadth and width are sometimes used interchangeably in the literature. When used in the context of layered circuits, usually inputs are not counted. Note that this corresponds to our definition of breadth.

We start with two special cases. For synchronous circuits we have the following definition.

Definition 8.1.1. Width of a synchronous circuit Let C be a synchronous circuit. Then the *width* of C is the maximum number of gates in the same level.

Similarly, we define the breadth of a layered circuit.

Definition 8.1.2. Breadth of a layered circuit Let C be a layered circuit with one output. Then the *breadth* of C is the maximum number of gates in the same layer.

Equivalently, the width of a layered circuit C is the maximum number of gates of the same depth, and the breadth of C is the maximum number of gates of the same height. See Definition 3.1.2. Also note that we do not count circuit inputs in width and breadth.

Pippenger [106] gave the following definition for the width of general circuits.

Definition 8.1.3. [106] **Thickness and width** Let C be a Boolean circuit of depth d with one output. The *thickness* of the dth level is 1. For $1 \leq l \leq d-1$, the *thickness* at level l is the number of gates at levels not exceeding l upon which one or more gates at levels exceeding l depend (as immediate predecessors). The *width* of C, denoted by $W(C)$, is the maximum thicknesses over all of its levels. The *width* of a Boolean function f, denoted by $W(f)$, is the minimum width over all circuits computing f.

Similarly, we generalize *breadth* for general circuits.

Definition 8.1.4. Broadness and breadth Let C be a Boolean circuit of depth d with one output. The *broadness* of the 0th layer is 1. For $1 \leq i \leq d-1$, the *broadness* of the ith layer is the number of gates at layers at least i upon which one or more gates at layers less than i depend (as immediate predecessors). The *breadth* of a Boolean circuit C, denoted by $B(C)$, is the maximum broadness over all layers. The *breadth* of a Boolean function f, denoted by $B(f)$, is the minimum breadth over all circuits computing f.

We define broadness and breadth only for circuits with one output, since height cannot be defined for multiple-output circuits. As in Definition 8.1.1 and

Definition 8.1.2, we do not count circuit inputs when computing thickness and broadness. Also notice that the numbering of levels and layers are different. For circuits with one output, the output is always in the dth level, where d is the depth of the circuit, whereas the output is always in the 0th layer.

Lemma 8.1.5. *In any circuit with one output, the thickness of each level is at least the number of gates in that level. In synchronous circuits with one output, the thickness of each level is equal to the number of nodes in that level.*

Proof. Consider the ith level. If $i = d$, where d is the depth of the circuit, then the thickness of the ith level is 1 by Definition 8.1.3. Suppose $1 \leq i \leq d-1$. In a circuit with one output, every gate in the ith level is used by at least one gate in the $(i+1)$th level. Then the claim follows from Definition 8.1.3.

Now consider a synchronous circuit C with one output. Since C is synchronous, every wire is between adjacent levels by Lemma 3.1.4. Let d be the depth of C. Our claim is true for the dth level, since the number of gates and the thickness of the dth level are both 1. Now let $1 \leq i \leq d-1$. Then for the ith level, the only gates at levels i or less that are used (as immediate predecessors) by gates at levels $i+1$ or more are exactly those gates at level i. So the thickness of the ith level is the number of gates in the ith level. □

The following corollary shows that for synchronous circuits, the two definitions of width in Definition 8.1.1 and Definition 8.1.3 are equivalent.

Corollary 8.1.6. *Let C be a synchronous circuit with one output. Let w_1 and w_2 be the widths defined in Definition 8.1.1 and Definition 8.1.3, respectively. Then $w_1 = w_2$.*

Proof. Follows from Lemma 8.1.5, since w_1 is the maximum number of gates in any level, and w_2 is the maximum thickness of any level. □

Lemma 8.1.7. *In any circuit with one output, the broadness of each layer is at least the number of gates in that layer. In layered circuits with one output, the broadness of each layer is equal to the number of nodes in that layer.*

Proof. Consider the ith layer. If $i = 0$, then the broadness of the ith layer is 1 by Definition 8.1.4. Suppose $1 \leq i \leq d-1$, where d is the depth of the circuit. By

Definition 3.1.1 and Definition 3.1.2, every gate in the ith layer is used by at least one gate in the $(i-1)$th layer. Then the claim follows from Definition 8.1.4.

Now consider a layered circuit C with one output. Since C is layered, every wire is between adjacent layers by Lemma 3.1.6. Let d be the depth of C. Our claim is true for the 0th layer, since the number of gates and the broadness of the 0th layer are both 1. Let $1 \leq i \leq d-1$. Then for the ith layer, the only gates at layers i or more that are used (as immediate predecessors) by gates at layers less than i are exactly those gates at layer i. So the broadness of the ith layer is the number of gates in the ith layer. □

The following corollary implies that for layered circuits, the two definitions of breadth in Definition 8.1.2 and Definition 8.1.4 are equivalent.

Corollary 8.1.8. *Let C be a layered circuit with one output. Let b_1 and b_2 be the breadths defined in Definition 8.1.2 and Definition 8.1.4, respectively. Then $b_1 = b_2$.*

Proof. Follows from Lemma 8.1.7, since b_1 is the maximum number of gates in any layer, and b_2 is the maximum broadness of any layer. □

In general, the width and breadth could be different for a fixed circuit. Figure 8.1 is a simple circuit whose width is linear in the input length but the breadth is a constant (the omitted gate type is not important for our purpose). This is the main motivation for the definition of breadth, since for example we can use the smaller of the two measures for a pebbling strategy.

However, the following lemma shows that the width equals the breadth if the given circuit is synchronous.

Lemma 8.1.9. *Let C be a synchronous circuit with one output. Let w and b be the width and breadth of C, respectively. Then we have $w = b$.*

Proof. We shall prove that the ith level of C is the $(d-i)$th layer of C, where d is the depth of C. Since for synchronous circuits, the width and breadth are the maximum size of levels and layers, respectively, the lemma follows.

Let l_i and λ_i be the ith level and layer of C, respectively. Let g be any node in l_i. Since C is synchronous, C is also layered by Lemma 3.1.7. Then every path from any input to g has length i, and every path from any input to the output has

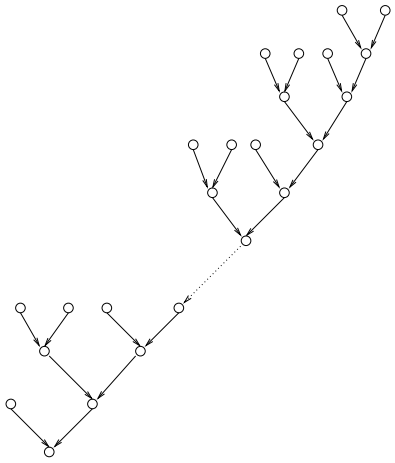

Figure 8.1: A circuit with constant breadth but linear width

length d. So every path from g to the output has length $d - i$. Therefore $g \in \lambda_{d-i}$, and we have $l_i \subseteq \lambda_{d-i}$.

Now let g be any node in λ_{d-i}. Since C is layered, every path from g to the output has length $d - i$. Also, since C is synchronous, every path from any input to the output has length d. So every path from any input to g must have length i, which means $g \in l_i$. So we have $\lambda_{d-i} \subseteq l_i$. □

A circuit is *internally synchronous* if any wire between two gates are between adjacent levels. An internally synchronous circuit may not be synchronous, since we could have a wire from an input to a gate of depth more than 1. The following lemma generalizes Lemma 8.1.9, that the breadth and the width are the same if the circuit is internally synchronous.

Lemma 8.1.10. *Let C be a Boolean circuit with one output of width w and breadth b. If for every wire (g, h) between two gates g, h, g and h belong to adjacent levels, then $w = b$.*

Proof. Let d be the depth of C. Let G be the DAG obtained from C if we remove all input nodes whose height is not d. So every source node (node with in-degree 0) in G has height d. Notice that the definitions of width and breadth are well-defined for any DAG, so let $B(G)$ and $W(G)$ be the breadth and width of G, respectively. Since we do not count input nodes when calculating the thickness and broadness of

C, we have $B(C) = B(G)$ and $W(C) = W(G)$. Then by similar arguments in the proof for Lemma 8.1.9, we have $w = b$. □

The following theorem shows that the breadth is always at most the width of a given circuit.

Theorem 8.1.11. *Let C be a Boolean circuit with one output. Then $B(C) \leq W(C)$.*

Proof. Let $l_0, \ldots, l_d$ be the levels of C. If every wire connecting two gates is between adjacent levels, then $B(C) = W(C)$ by Lemma 8.1.9. So let (g, h) be a wire such that g and h are both gates, and g, h belong to the ith and jth level, respectively, where $j > i + 1$. Observe that for each level in $l_i, l_{i+1}, \ldots, l_{j-1}$, (g, h) contributes exactly one to the thickness of that level. So if we interpolate (g, h) by adding $j - i - 1$ copy gates ($\vee$ gate with two same inputs, for example), then we do not increase the thickness of the levels.

Apply this interpolation for every wire between non-adjacent levels. Let C' be the circuit obtained. Since the thickness of levels does not change, we have $W(C') = W(C)$. We only add gates, the broadness cannot decrease. Since the broadness of layers cannot decrease, we have $B(C') \geq B(C)$. By Lemma 8.1.9, we have $B(C) \leq B(C') = W(C') = W(C)$. □

Corollary 8.1.12. *For any Boolean function $f : \{0,1\}^n \to \{0,1\}$, we have $B(f) \leq W(f)$.*

The following lemma relates the total fanout of the gates in a level to the width of the circuit.

Lemma 8.1.13. *Let l_i be the ith level of a Boolean circuit C with one output. Then the sum of the fanouts of gates in l_i is at most triple the width of C. In particular,*

$$\sum_{g \in l_i} fanout(g) \leq 3W(C).$$

Proof. Let $E_i = \{(g, h) : \ g \in l_i\}$ be the set of all wires originating from l_i. Then E_i is the disjoint union of

$$E_{i1} = \{(g, h) : \ g \in l_i, h \in l_{i+1}\}$$

and

$$E_{i2} = \{(g,h) : \ g \in l_i, h \in l_j, j > i+1\}.$$

For E_{i1}, since each gate $h \in l_{i+1}$ has fanin 2, we have $|E_{i1}| \leq 2|l_{i+1}| \leq 2W(C)$ by Lemma 8.1.5.

Now consider E_{i2}. Let k be

$$k = \min\{j : \exists (g,h), g \in l_i, h \in l_j, j > i+1\}.$$

So the kth level is the closest level to the ith level such that there is a "jump" between the two levels. Since every wire in E_{i2} connects a gate in l_i to the kth level or higher, the width of the $(k-1)$th level is at least $|E_{i2}|$. Then we have

$$|E_i| = |E_{i1}| + |E_{i2}| \leq 3W(C).$$

□

Given a graph G and an embedding of G into the plane, a *crossing* happens when an edge intersects another edge in this embedding. Intuitively, the number of crossings is a good measure to see if a graph is "close" to planar. The following lemma formalizes the intuition. See Wegener [146] for the proof.

Lemma 8.1.14. *(Credited to McColl in Wegener [146]) Let C be a Boolean circuit of size s such that C has c crossings in some embedding. Then there exists a planar circuit C_p simulating C such that the size of C_p is at most $s + 3c$.*

The following theorem relates planarity to the breadth and width of a circuit.

Theorem 8.1.15. *Let C be a Boolean circuit with one output of size s, breadth b, and width w. Let t be the size of the maximum planar subgraph of C. Then there is an embedding such that the number of crossings is at most $O((s-t)\min(b,w)) = O((s-t)b)$.*

Proof. We first prove that the number of crossings is $O((s-t)w)$. The proof for $O((s-t)b)$ is similar. The theorem then follows from Theorem 8.1.11.

Let H be any planar subgraph of C of size t, so that adding any edge to H will yield crossings. Let l_i be the ith level in C. Let (g_1, g_2) be any edge in C but

not in H, and let g_1 and g_2 be in the ith and jth level, respectively. Then there are two kinds of wires (g_1, g_2) might intersect.

1. Wires (h_1, h_2) such that $h_1 \in l_p$ and $h_2 \in l_q$, where $p \leq i$ and $i+1 \leq q \leq j$. Then the number of such crossings is at most the thickness of l_i.

2. Wires (h_1, h_2) such that $h_1 \in l_p$ and $h_2 \in l_q$, where $i \leq p \leq j-1$ and $q \geq j$. Then the number of such crossings is at most the thickness of l_{j-1}.

It is easy to see that there is an embedding in which (g_1, g_2) does not intersect wires (h_1, h_2) such that $h_1 \in l_p$ and $h_2 \in l_q$, where $p < i$ and $q > j$, or $p > i$ and $q < j$, since (g_1, g_2) either "encloses" (h_1, h_2) or vice versa.

So the wire (g_1, g_2) will intersect at most $2w$ wires. The number of such (g_1, g_2) is at most $s - t$, so the number of crossings is $O((s-t)w)$. □

8.2 The Relationships among Breadth, Width, Space, and Pebble Games

Our goal in this section is to relate various computational resources. In particular, we will be interested in resources related to the space complexity, including the breadth and width of a circuit defined in Section 8.1, and the pebble games defined in Section 2.1. We will use the fact that languages decided in simultaneous time $t(n)$ and space $s(n)$ can be decided by circuit families of size $O(t(n)\log(n+s(n)))$. This result was given by Schnorr [124] based on the result by Pippenger and Fischer [103]. Also see Theorem 8.3.1 and Theorem 8.3.9 in later sections for the construction.

The following theorem says that the breadth and the width of a Boolean function cannot be too big in terms of its space complexity and input length. It also shows an interesting symmetry between the width and breadth of Boolean functions.

Theorem 8.2.1. *Let $f : \{0,1\}^n \to \{0,1\}$, and let $s(n) \geq \log n$ be the space complexity of f on Turing machines. Then we have $B(f) = O(\min(n, s(n)))$ and $W(f) = O(\max(n, s(n)))$.*

Proof. We shall assume familiarity with Cook's table method [24] for translating the computations of Turing machines into Boolean circuits. See Chapter 8 in Papadimitriou [100] for a recent treatment of this topic.

$\mathbf{B(f) = O(n)}$ Construct a formula F by any DNF of f. For each conjunction, we construct the most unbalanced formula for the conjunction (i.e. each gate takes at least one input bit). Then we construct another similar unbalanced formula for the disjunction, where the inputs in this chain are the formulas for the conjunctions. Then the resulting formula has breadth $O(n)$. Notice that the width of F can be very large. If each conjunction has the same size, then the width will be the size of the disjunction, which can be as large as $\Omega(2^n)$.

$\mathbf{B(f) = O(s(n))}$ Our construction involves the storage-access function. It is also known as the addressing function. This is a well-studied function. We give a special circuit implementation of the storage-access function.

Lemma 8.2.2. *Given $x = x_1...x_n$ and $1 \leq k \leq n$, there exists a circuit of $O(\log n)$ breadth that outputs x_k.*

Proof. Notice that x_k can be computed by the following expression,

$$x_k = \vee_{i=1}^{n} EQ(i,k) \wedge x_i,$$

where $EQ(i,k)$ outputs true if i equals k, and false otherwise. Note that $EQ(i,k)$ can be computed by a circuit of $O(\log n)$ breadth, by considering the most unbalanced tree-like circuit comparing each bit of i and k. To compute x_k in $O(\log n)$ breadth, simply put a suitable number of dummy gates to separate each subcircuit computing EQ. We also use dummy gates to copy the $O(\log n)$ bits representing k to be accessible to the next EQ circuit. Thus x_k can be computed by a circuit of breadth $O(\log n)$. □

We now show that $B(f) = O(s(n))$. Let M be a deterministic Turing machine with space complexity $s(n)$. It is easy to see that M can be modified such that in each computation step, M also writes the location of its input head on the work tape. Let M' be this machine modified from M. Then M' uses $O(s(n) + \log n) = O(s(n))$ space. Now we apply the table method on M', but on its work tapes only. We use Lemma 8.2.2 to compute the input bit used in each step. To keep the breadth of M' small, we use a suitable number of dummy gates to separate each subcircuit corresponding to each computation step of M'. Thus the breadth of the final circuit is $O(s(n) + \log n) = O(s(n))$ for $s(n) \geq \log n$.

$\mathbf{W(f) = O(\max(n, s(n)))}$ Let M be a deterministic Turing machine with space complexity $s(n)$. Let C be the Boolean circuit obtained by applying the classical table method to M. We add copy gates to C such that there are no wires between non-adjacent levels. Then the resulting circuit is synchronous, and the width is linear in the number of columns in the table, which is $O(n + s(n))$. So we have $W(f) = O(n + s(n)) = O(\max(n, s(n)))$. □

Notice that the width of a circuit is not necessarily upper-bounded by the space complexity of the function computed by the circuit. An example is Figure 8.1. It can be computed in log space, but the width of the circuit is linear in the input length. Also notice that by Borodin's theorem [11] (Theorem 1.2.1), we can simulate a depth d circuit by a space $O(d)$ Turing machine. This implies $B(f) = O(D(f))$ under sufficient uniformity assumptions.

The following theorem says that if the circuit has small breadth, then we have an efficient one-person pebbling strategy.

Theorem 8.2.3. *Let C be a Boolean circuit with one output of breadth b. Let p_1 be the minimum number of pebbles used in the one-person pebble games. Then we have $p_1 \leq 2b + 2$.*

Proof. Since we do not count inputs with respect to the broadness of layers or the thickness of levels, we cannot pebble all the input nodes at the beginning of the game. Instead, we only pebble the input nodes when necessary, and remove the pebbles on the input nodes as soon as they are not needed.

Let d be the depth of the circuit C. The pebbling strategy is defined as follows.

1. For a gate g in the $(d-1)$th layer, we pebble g by first pebbling the inputs of g (which must be inputs of C), pebble g, and then remove the pebbles on the inputs of g. Repeat until every gate in the $(d-1)$th layer is pebbled.

2. For each i from $d-2$ to 0, the strategy has two phases. In the first phase, we pebble the gates in the ith layer. If a gate g to be pebbled has an input that is an input of C, then we first pebble that input, and remove the pebble on the input as soon as g is pebbled. In the second phase, we remove pebbles on gates that are not used by gates in layers less than i. While each gate

belongs to exactly one layer, a gate may be counted towards the breadth of several layers. If a pebbled gate is used by gates in layers less than i, then we need the gate to remain pebbled in order to pebble gates in layers less than i. Therefore we do not remove pebbles on these gates in the second phase.

For the $(d-1)$th layer, the strategy uses at most $b+2$ pebbles. Now consider layers from $d-2$ to 0. Note that during the first phase, we have at most $2b+2$ pebbles on C. After the second phase, the pebbled gates are exactly those gates at layers at least i that are used by gates at layers less than i. That is, the number of pebbled gates is the broadness of the ith layer, which is at most b. Thus we have $p_1 \leq 2b+2$. □

The following theorem says that if the circuit has small breadth, then we have an efficient two-person pebbling strategy.

Theorem 8.2.4. *Let C be a Boolean circuit with one output of breadth b and depth d. Let p_2 be the minimum number of pebbles used in the two-person pebble games. Then we have $p_2 = O(b \log \frac{d}{b})$ if $b < d/2$, and $p_2 = O(b)$ otherwise.*

Proof. Suppose that $b \geq d/2$, then in each round the pebbler simply pebbles the two immediate predecessor of the currently challenged node. Then we have $p_2 = O(b)$.

Now suppose that $b < d/2$. For $1 \leq i \leq d-1$, define the *cut* L_i to be the set of gates in layers at least i that are used by at least one gate in layers less than i. We also define L_0 to be the set consisting of the output gate. Then by Definition 8.1.4, $|L_i|$ is the broadness of the ith layer. It is easy to see that every path from a gate in a layer at least i to a gate in a layer less than i must contain a gate in L_i, which is why we refer to the sets L_i as cuts. Note that the sets L_i may overlap. Define the j-th block of C to be

$$B_j = \{L_{jx}, L_{jx+1}, \ldots, L_{(j+1)x-1}\}.$$

That is, B_j consists of the consecutive cuts L_{jx} to $L_{(j+1)x-1}$, inclusively. Now we define a two-person pebbling strategy as follows. The strategy has two phases. In the first round of the first phase, the challenger challenges the output of C, which is in the first cut of the first block B_0. Then the pebbler pebbles the first cut of the middle block, i.e. the first cut L_{mx} in B_m, where $m = \frac{1}{2}\lceil \frac{d}{x} \rceil$. If the challenger

stays, then the pebbler pebbles the first cut of the middle block between B_0 and B_m. Otherwise the pebbler pebbles the first cut of the middle block between B_m and $B_{\lceil \frac{d}{x} \rceil}$. The two players proceeds as above until the first cut of the block closest to the challenger is pebbled, that is until the challenger is located in a block B_j such that the first cut of the block B_{j+1} is pebbled. Then the game enters the second phase. In each round of the second phase, the pebbler simply pebbles the two immediate predecessors of the currently challenged node. It is easy to see that the pebbler will eventually win the game.

Now we count the number of pebbles used. In the first phase, since there are at most $\log \lceil \frac{d}{x} \rceil$ rounds and b pebbles are used in each round, the number of pebbles used in the first phase is $O(b \log \lceil \frac{d}{x} \rceil)$. For the second phase, the pebbler uses $2x$ pebbles. Thus the total number of pebbles used is

$$O(b \log \lceil \frac{d}{x} \rceil) + 2x = O(b \log \frac{d}{b})$$

if we let $x = b$. □

Intuitively, a Turing machine M can evaluate a uniform circuit level-by-level or layer-by-layer, and the number of gate values M to remember is at most the width or the breadth multiplied by the log of the circuit size (to identify a node). However, it is slightly more complicated to implement this idea within the given space bound.

Theorem 8.2.5. *Let L be a language decided by a uniform circuit family, and let C_n be the Boolean circuit with one output for L on inputs of length n. Let $h(n)$ be the space complexity to generate the description of C_n together with the height of each gate. Let $b(n)$, $d(n)$, and $z(n)$ be the breadth, depth, and size of C_n, respectively. Let $s(n)$ be the space complexity of L. Then we have*

$$s(n) = O(h(n) + \min(d(n), b(n) \log z(n))).$$

We already have $s(n) = O(h(n) + d(n))$ by Theorem 1.2.1, so we only need to show $s(n) = O(h(n) + b(n) \log z(n))$.

Proof. Let $l(i)$ be the ith layer of C_n, and let $\lambda(i)$ be the gates contributing to the broadness of the ith layer. So we have $l(0) = \lambda(0)$ (the output).

First we describe how the Turing machine M utilizes its work tapes when evaluating a node $g \in C$. The evaluation of C_n can be done without writing the whole circuit description of C_n on the tape, by the standard space-bounded composition of functions. Then the total space required is the space for the evaluation, plus the space to generate the circuit description.

If g is an input node, then M searches for the value on the input tape. If g is a gate, then M searches for g's predecessors on the work tape (assuming the values of g's predecessors are already there) or on the input tape (if the predecessor of g is an input node), and computes g accordingly. In both cases, once M has g's value, M will write down g's record. The record of g is defined as the name of g, the height of g, and the value of g, which uses $\log z(n)$, $\log d(n) \leq \log z(n)$, and 1 bits, respectively. So we need $2 \log z(n) + 1$ bits to store the record of g.

Now we describe how M evaluates C_n. Given $0 \leq i \leq d(n) - 1$, we define a machine M' to generate $l(i)$ as follows. M' uses space $h(n)$ to generate the description of C_n and the height of each gate, and outputs gates whose height is i.

In the beginning M uses M' to generate $l(d(n)-1)$, which is the $(d(n)-1)$th layer. Since the two inputs to each gate g in the $(d(n)-1)$th layer are inputs of C_n, M can look up their values and compute the value of g. M writes down the records of all gates in the $(d(n)-1)$th layer on the work tape. Next for i from $d(n)-2$ to 0, M uses M' to generate $l(i)$, evaluates gates in $l(i)$ using records on the work tape, and writes down the records of gates in $l(i)$. M then checks for each gate $g \in l(j)$ whose record is on the work tape, where $j \geq i+1$. If g has a parent in $l(k)$ where $k < i$, then M keeps the record of g on the work tape. Otherwise suppose that every parent of g is in $l(k)$ where $k \geq i$. Then M deletes the record of g on the work tape, since the record of g will not be used in subsequent computations. It is easy to see that after M has processed every such gate g, the number of records on the work tape is the broadness of $l(i)$.

For the space used, each record uses $O(\log z(n))$ bits, and there are at most $b(n)$ records. Thus M uses $O(h(n) + b(n) \log z(n))$ bits, where $h(n)$ is used to generate the description of C_n and the height of each gate. □

We write $DTiSp(t(n), s(n))$ to denote the class of languages decided by de-

terministic Turing machines in time $t(n)$ and space $s(n)$ on inputs of length n. Then Theorem 8.2.5 implies the following corollary.

Corollary 8.2.6. *Let L be a language decided by a uniform circuit family, and let C_n be the Boolean circuit with one output for L on inputs of length n. Let $t(n)$ and $h(n)$ be the time and space, respectively, to generate the description of C_n as well as the height of each gate. Let $z(n)$ and $b(n)$ be the size and breadth of C_n, respectively. Then we have*

$$L \in DTiSp(b(n)t(n)z(n) + b(n)z^2(n)\log z(n), h(n) + b(n)\log z(n)).$$

Proof. We show that the machine M in Theorem 8.2.5 runs in time $O(b(n)t(n)z(n)+ b(n)z^2(n)\log z(n))$ and space $O(h(n)+b(n)\log z(n))$. The space bound follows from Theorem 8.2.5. Let $d(n) \leq z(n)$ be the depth of C_n. For the time bound, first observe that generating $l(i)$ takes $O(t(n)+z(n)\log d(n))$ time, where $O(z(n)\log d(n))$ is used to check if the gates has height $i \leq d(n)$. Evaluating one gate in $l(i)$ takes $O(b(n)\log z(n))$ time, since we may need to scan the whole work tape to retrieve the values of its two children, and M uses $O(b(n)\log z(n))$ space on the work tape. So evaluating all gates in $l(i)$ takes $O(b^2(n)\log z(n))$ time, since we have $|l(i)| \leq b(n)$ by Lemma 8.1.7. Given a gate $g \in l(i)$, it takes $O(t(n)+z(n)\log d(n))$ time to check the heights of all parents of g, where $\log d(n)$ time is used to check the height of each gate. Then checking the parents of all gates whose records is on the work tape takes $O(b(n)(t(n) + z(n)\log d(n)))$ time. Since there are $d(n)$ layers, the total time is

$$\begin{aligned} & d(n)O(t(n) + z(n)\log d(n) + b^2(n)\log z(n) + b(n)(t(n) + z(n)\log d(n))) \\ = \; & d(n)O(b(n)t(n) + b(n)z(n)\log z(n)) \\ = \; & O(b(n)t(n)z(n) + b(n)z^2(n)\log z(n)). \end{aligned}$$

□

We conclude this section with the relationship between the one person pebble game and space complexity. We start with the direction of simulating space-bounded computations by the one person pebble game.

Corollary 8.2.7. *Let L be any language in $SPACE(s(n))$, where $s(n) \geq \log n$. Then there exists a circuit family with one output for L such that the one-person pebble game on C uses $O(s(n))$ pebbles, where C is a circuit with input length n.*

Proof. By Theorem 8.2.1, we know that there exists a circuit family for L such that each circuit has breadth $s(n)$. The claim then follows from Theorem 8.2.3. □

For the direction of simulating the one-person pebble game by space-bounded computations, one concern is that in the pebble game, the pebbler does not have any computational limitation. This raises problems when we try to evaluate the circuit by the pebbling strategy. For example, if the pebbler's strategy is to pebble the smallest separator and proceed recursively, then it is unlikely that we can use this strategy to find a polynomial time algorithm to evaluate the circuit. This is because finding the minimum graph separator in general is NP-hard (Hyafil and Rivest [61], also see Chapter 3 of Rosenberg and Heath [117]). So the existence of the pebbling strategy does not necessarily mean the strategy can be computed by a Turing machine. This motivates the following definition.

Definition 8.2.8. Given a Boolean circuit C with one output, a *pebbling strategy* is a sequence of moves, where each move is represented by $1 + \log |C|$ bits. The $\log |C|$ bits are used to denote a gate, and the extra one bit is used to denote placing or removing a pebble on that gate. We say that a pebbling strategy on C is $h(n)$-*uniform* if there exists a deterministic Turing machine M such that, on input C, the pebbling strategy is written on the output tape after M halts. Furthermore, M uses $O(h(n))$ space on the work tapes.

The notion of a $h(n)$-uniform pebbling strategy does not impose great restrictions when design strategies in practice. In fact, the pebbling strategies in [25], [102], [59] (one-person pebble games), and [34], [141] (two-person pebble games) are all uniform.

Theorem 8.2.9. *Let L be a language decided by an $h_1(n)$-uniform Boolean circuit family with one output, and let C_n be the circuit of size $z(n)$ in the family for inputs of length n. Let $p_1(n)$ be the minimum number of pebbles to pebble the output of C_n by an $h_2(n)$-uniform strategy in the one-person pebble game. Then $L \in SPACE(s(n))$, where $s(n) = O(h_1(n) + h_2(n) + p_1(n) \log z(n))$.*

Proof. Let M_p be the Turing machine computing the pebbling strategy. We shall design a Turing machine M such that M computes f in space $O(h_1(n) + h_2(n) + p_1(n) \log z(n))$. Since C_n is uniform, M first runs M_p and feeds the description of C_n to M_p when necessary. M pauses M_p as soon as M_p writes a move on M_p's output tape (which is simply another work tape of M). Then the Turing machine M computes f according to the move as follows:

- If the move places a pebble on a node, then M computes the value of that node and writes down the record for that node on its tape, where the record consists of the name and the value of the node.
- If the move removes a pebble on a node, then M deletes the record for that node and rearranges the records such that the records of all nodes are contiguous.

By the rules of the pebble game, the pebbler can only put a pebble on a gate when the gate's predecessors both have pebbles on them. So M can simply scan its work tape for the values of the predecessors, and compute the gate's value accordingly. Also, after M deletes a record, M needs to put the remaining records together, so that the records are not scattered over the work tape.

For the space complexity of M, the record for each node needs $1 + \log z(n)$ bits, where $\log z(n)$ bits are used for the name of the node, and 1 bit is used to store the value of that node. Therefore at most $p_1(1 + \log z(n))$ bits are used on the work tape for the strategy. Thus the space complexity is $h_1(n) + h_2(n) + p_1(n)(1 + \log z(n)) = O(h_1(n) + h_2(n) + p_1(n) \log z(n))$. □

8.3 Simultaneous Time and Space of Input-Oblivious Turing Machines

8.3.1 The Construction of Oblivious Turing Machines

The following theorem was given by Pippenger and Fischer [103], which simulates a Turing machine by a family of Boolean circuits.

Theorem 8.3.1. *[103] Let L be a language with time complexity $t(n)$ on a multi-tape Turing machine. Then there exists a family of circuits for L with size $O(t(n) \log t(n))$.*

To prove the above theorem, Pippenger and Fischer used the notion of oblivious Turing machines, which was introduced by Fischer, Meyer, and Rosenberg [37].

Definition 8.3.2. [37] A Turing machine is called *oblivious* if for any given tape, the sequence of its head moves is the same for all inputs of the same length.

Pippenger and Fischer [103] showed that we can simulate a multi-tape Turing machine by an oblivious Turing machine without too much cost in running time.

Theorem 8.3.3. *[103] Let M be a Turing machine with one read-only input tape, k work tapes, and one output tape. Let $t(n) \geq n$ be the time complexity of M on inputs of length n. Then there exists an oblivious Turing machine M' with one read-only input tape, two work tapes, and one output tape such that M' simulates M with time complexity $O(t(n) \log t(n))$.*

Since our results in the following sections heavily depend on Fischer and Pippenger's result, we need to describe their construction in details. This is done in the following two lemmas. In the first lemma, we design a Turing machine M_1 simulating M such that M_1 is oblivious only on its input and output tapes, and may or may not be oblivious on its work tapes. In the second lemma, we design a Turing machine M_2 simulating M_1 such that M_2 is oblivious on all its work tapes, and behaves exactly the same as M_1 on the input and output tapes. Therefore the oblivious Turing machine M_2 simulates M. The proof for their correctness can be found in Pippenger and Fischer [103], Wegener [146] and Vollmer [143].

We begin by the following definition.

Definition 8.3.4. Let f be a function from non-negative integers to non-negative integers. We say that f is a *proper complexity function* if f is non-decreasing, and there exists a Turing machine M such that on input x, M writes $1^{f(|x|)}$ ($f(x)$ number of 1's) to the output tape using time $O(|x| + f(|x|))$ and space $O(f(|x|))$. We will usually only consider proper complexity functions unless stated otherwise.

Lemma 8.3.5. *[103] Let M be a Turing machine with one read-only input tape, k work tapes, and one output tape. Let $t(n) \geq n$ be the time complexity used by M on inputs of length n such that $t(n)$ is a proper complexity function. Then there exists another Turing machine M_1 simulating M with running time $O(t(n))$, such that M_1 is oblivious only on its input and output tapes.*

Note that M_1 may or may not be oblivious on its work tapes.

Proof. Let $x \in \{0,1\}^n$ be the input, so the running time of M on x is $t(|x|)$ We will construct a Turing machine M_1 such that M_1 has one input tape, $k+3$ work tapes, one output tape, and M_1 is oblivious on its input and output tapes (may or may not be oblivious on the work tapes). In the beginning, M_1 copies the input x from its input tape to its first work tape. This can be done obliviously. Then M_1 simulates M for $t(|x|)$ steps. Recall that $t()$ is a proper complexity function. Then this step can be done by running the Turing machine witnessing $t(|x|)$ on the second work tape of M_1, and treats the first and third work tapes as the input and output tape of M, respectively. M_1 uses the other k work tapes as if they were the k work tapes of M. Furthermore, we require that M_1 does not move the heads of its own input and output tapes during this simulation. After the simulation, the contents on M_1's third work tape are copied to the output tape of M_1. This can also be done obliviously. Therefore M_1 is oblivious on its input and output tapes. □

The space complexity of M_1 can be itemized as follows. The extra three work tapes of M_1 use space n, $t(n)$, and $u(n)$, respectively, where $u(n) \leq t(n)$ is the output length of M. The remaining k work tapes of M_1 use the same space as M's k work tapes.

Now we shall simulate M_1 by a Turing machine with work tapes infinite on both directions. This slight change in the machine model simplifies notations in the next section when we construct the corresponding family of circuits.

Lemma 8.3.6. *[103] Let M_1 be the Turing machine as in Lemma 8.3.5. That is, M_1 is oblivious on its input and output tapes. Let $t(n) \geq n$ be the time complexity of M_1 on inputs of length n. Then there exists a Turing machine M_2 with two-way-infinite work tapes such that, M_2 simulates M_1 with time complexity $O(t(n) \log t(n))$, M_2 has two work tapes, and M_2 is oblivious on all tapes.*

Proof. We shall describe the construction of M_2 but not the proof of correctness. See Pippenger and Fischer [103], Wegener [146], or Vollmer [143] for a complete proof.

Let Σ_1 be the alphabet of M_1. Since the input and output tapes of M_1 are already oblivious, we let M_2 move exactly the same as M_1 on the input and output tapes. The head movements of M_2's work tapes are defined as follows. The first

work tape of M_2 is infinite on both sides. We will partition the first work tape of M_2 into k channels, one channel for each work tape of M_1. Furthermore, each channel has three tracks, and each entry in a track could have any symbol in $\Sigma_1 \cup \{\#\}$, where $\#$ is a new symbol not in Σ_1. So each symbol on this work tape of M_2 is a tuple in $(\Sigma_1 \cup \{\#\})^{3k}$. Given a fixed channel c and $i \in 0, 1, 2$, the ith track in the cth channel is divided into blocks $B_{c,i,j}$, where

- $B_{c,i,0}$ contains position 0 of track i in channel c (of the first tape of M_2);
- if $j > 0$, then $B_{c,i,j}$ contains the 2^{j-1} positions $2^{j-1}, \ldots, 2^j - 1$;
- if $j < 0$, then $B_{c,i,j}$ contains the $2^{|j|-1}$ positions $-(2^{|j|} - 1), \ldots, -2^{|j|-1}$.

Note that if $j > 0$, then the size of $B_{c,i,j}$ is twice the size of $B_{c,i,j-1}$. If $j < 0$, then the size of $B_{c,i,j}$ is half the size of $B_{c,i,j-1}$.

Let the segment $S_{c,j}$ consist of the blocks $B_{c,0,j}$, $B_{c,1,j}$, and $B_{c,2,j}$. Note that $S_{c,0}$ contains partial contents of the 0th cell in the first work tape of M_2, $S_{c,1}$ contains partial contents of the first cell in the first work tape of M_2, $S_{c,2}$ contains partial contents of the second and third cells in the first work tape of M_2, $S_{c,3}$ contains partial contents of the fourth to the seventh cells in the first work tape of M_2, and so on. We classify blocks and segments into the following types. See Figure 8.2 for an example.

- A block is called *empty* if it only contains #'s.
- A block is called *full* if it does not contain any #'s.
- A segment is called *clean* if the segment has either 1 or 2 full blocks

Observe that at the beginning of the simulation, all blocks on channels except the first channel are either full or empty, and all segments on channels except the first channel are clean, since they correspond to the empty work tapes of M_1. Now consider the first channel, which corresponds to the input tape of M_1. Assume that the input length is a power of 2 (otherwise we can pad the input to the nearest power). For the first channel, since it contains the input x, we can make a one-time scan of the first channel in the beginning such that every block in the 0th track of the first channel is full, and every block in the other two tracks of the first channel is empty. That is, every segment in the first channel has exactly one full block.

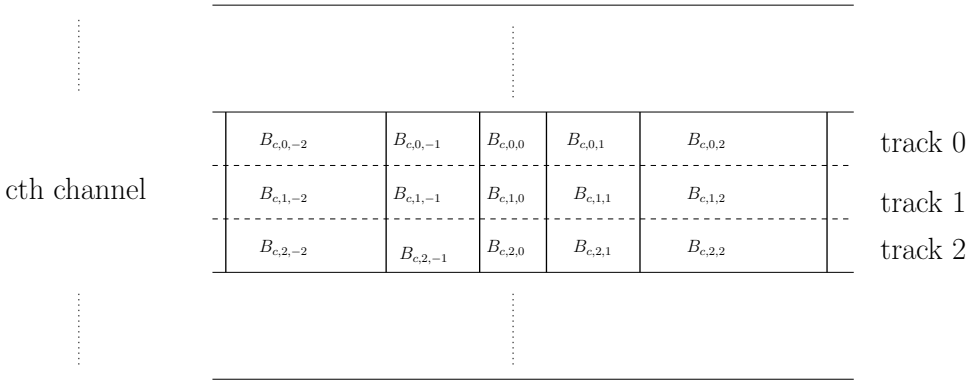

Figure 8.2: The first work tape of M_2

Our goal is to construct a program $sim(m)$ that simulates 2^m steps of M_1. Before we define $sim(m)$, we require that the following invariant holds after each $sim(m)$.

1. Each block is either full or empty.

2. The contents of the cth tape of M_1 after t computation steps can be read after t simulation steps by reading the full blocks $B_{c,i,j}$ according to the lexicographical order on the pairs (j, i).

3. The head of M_2's first work tape scans cell 0, which contains segments $S_{c,0}$ for all c. In the first track of each channel in this cell is the information that M_1 reads at that time.

4. During the 2^m simulation steps the head of M_2's first work tape only visits the segments $S_{c,-(m+1)}, \ldots, S_{c,m+1}$ for the cth channel. At the end of the simulation, $S_{c,-m}, \ldots, S_{c,m}$ are clean, and the number of full blocks of $S_{c,-(m+1)}$ and $S_{c,m+1}$ is at most by 1 larger or smaller than before the simulation.

All conditions are satisfied at the beginning of the simulation. We claim that $sim(0)$ can be performed obliviously by the following argument. M_2 knows the state q of M_1 and reads in the first track of each channel the same information $a_0, \ldots, a_{k-1}$ as M_1 reads in. Let $\delta(q, a_0, \ldots, a_{k-1}) = (q', b_0, \ldots, b_{k-1}, d)$, where δ is the transition function of M_1, and d is either L (left), R (right), or N (no move).

M_2 remembers q' in its finite control, and replaces $B_{c,0,0}$ by b_c. Now we have two cases.

1. If $d = L$, then the letters in $B_{c,0,0}$ and $B_{c,1,0}$ are transferred to $B_{c,1,0}$ and $B_{c,2,0}$, respectively. The letter in the last full block of $S_{c,-1}$ is then transferred to $B_{c,0,0}$. Now suppose that $B_{c,2,0}$ becomes full, then the letters in $B_{c,0,1}$ and $B_{c,1,1}$ are transferred to $B_{c,1,1}$ and $B_{c,2,1}$, respectively. The letter in $B_{c,2,0}$ is then transferred to $B_{c,0,1}$.

2. If $d = R$, then the letter in $B_{c,0,0}$ is transferred to the last empty block of $S_{c,-1}$, and the letters in $B_{c,1,0}$ and $B_{c,2,0}$ are transferred to $B_{c,0,0}$ and $B_{c,1,0}$, respectively. Suppose now that $B_{c,0,0}$ becomes empty, then the two letters in $B_{c,0,1}$ are transferred to $B_{c,0,0}$ and $B_{c,1,0}$. The letters in $B_{c,1,1}$ and $B_{c,2,1}$ are then transferred to $B_{c,0,1}$ and $B_{c,1,1}$, respectively.

Notice that all operations are possible because by assumption $S_{c,-1}$, $S_{c,0}$, and $S_{c,1}$ are all clean in the beginning. Since M_2 does not know d in advance, M_2 performs all above operations, but only actually writes down the correct symbols. After the operations are done, M_2 moves the head back to position 0.

We define $sim(m)$ for $m > 0$ by

$$sim(m) := sim(m-1); clean(m); sim(m-1); clean(m),$$

where $clean(m)$ is used to make the segments $S_{c,-m}$ and $S_{c,m}$ clean for all channels and is defined as follows. To clean $S_{c,m}$, the head moves to the right until it reaches the first cell of $S_{c,m}$ (this can be done using a counter on another work tape). There are three cases.

1. $S_{c,m}$ is clean. Then M_2 does not do anything.

2. All blocks of $S_{c,m}$ are empty. Then the first block of $S_{c,m+1}$ is broken into two pieces and transferred into the first two blocks of $S_{c,m}$, and the information in $B_{c,1,m+1}$ and $B_{c,2,m+1}$ are transferred into $B_{c,0,m+1}$ and $B_{c,1,m+1}$, respectively.

3. All blocks of $S_{c,m}$ are full. Then the last two full blocks of $S_{c,m}$ are concatenated and transferred into $B_{c,0,m+1}$, and the original information in $B_{c,0,m+1}$ and $B_{c,1,m+1}$ are transferred into $B_{c,1,m+1}$ and $B_{c,2,m+1}$, respectively.

We apply similar procedures to clean $S_{c,-m}$.

Notice that M_2 can determine which case is true by just checking a single cell of $S_{c,m}$. Since M_2 does not know which case is true in advance, M_2 performs all cases but only writes down the correct symbols. By the recursion tree of $sim(m)$, $sim(m)$ consists of 2^m $sim(0)$ simulation steps, and the tth $sim(0)$ is followed by $clean(1), \ldots, clean(l)$, where

$$l = \begin{cases} \max\{r|\ 2^{r-1} \text{ divides } t\} & \text{if } t \neq 2^m \\ m & \text{if } t = 2^m \end{cases} \tag{8.1}$$

Since $sim(0)$ and $clean(m)$ can be implemented obliviously, the program $sim(\lceil \log t(n) \rceil)$ is the oblivious Turing machine M_2 simulating M_1. □

Lemma 8.3.7. *[103] Let M, M', and $t(n)$ be defined as in Theorem 8.3.3, and let $s(n) \geq \log n$ be the space used by M. Then the work tapes of M' is utilized as follows.*

- *On the first work tape, M' uses $O(n)$, $O(t(n))$, and $O(t(n))$ spaces in the first three channels for the input, witnessing Turing machine (yardstick), and output, respectively. M' uses $O(s(n))$ for the remaining channels on the first work tape.*
- *On the second work tape, M' uses at most $O(s(n) + \log s(n))$ cells, where $O(s(n))$ is used to transfer information between blocks in the clean(m) program, and $\log s(n)$ is used for the counter.*

Proof. Follows immediately from the proofs for Lemma 8.3.5 and Lemma 8.3.6. □

The following property of oblivious Turing machines will be useful later.

Lemma 8.3.8. *[143] Let M, M', and $t(n)$ be defined as in Theorem 8.3.3, and let $s(n) \geq \log n$ be the space used by M. Let $k \leq \log s(n)$ be any positive integer. Then M' calls the subprogram clean(k) for $t(n)/2^{k-1}$ number of times.*

Proof. Without loss of generality, assume that $t(n)$ is a power of 2. By Equation 8.1, on input x, M' calls $clean(k)$ at the following times:

$$2^{k-1}, 2 \cdot 2^{k-1}, 3 \cdot 2^{k-1}, \cdots, t(|x|).$$

Then the number of times that M' calls $clean(k)$ is $t(|x|)/2^{k-1}$. □

8.3.2 Circuit Families from Oblivious Turing Machines

The classical method to convert Turing machines to Boolean circuits is the table method (see [100] Chapter 8). We could try to apply the table method directly to the M' in Theorem 8.3.3. However, the problem is that in the table method, the size of the rows is at least $n + s(n)$. So the total size of the circuit is at least $t(n)(n+s(n)) \geq n^2$ if we assume the Turing machine needs at least linear time. This yields quadratic increase either in running time or in circuit size, and we need to avoid quadratic increase. In Theorem 8.3.11 we obtain a construction that overcomes this problem on input-oblivious Turing machines. Our construction is based on the following theorem of Schnorr [124]. We slightly modify the construction such that we obtain circuits of small breadth.

Theorem 8.3.9. *[124] Let L be a language with time complexity $t(n)$ and space complexity $s(n) \geq \log n$ on a deterministic multi-tape Turing machine. Then there exists a family of circuits for L such that each circuit has size $O(t(n)\log(n+s(n)))$.*

Definition 8.3.10. A Turing machine is called *input-oblivious* if for the input tape, the sequence of its head moves is the same for all inputs of the same length.

We now show how to simulate input-oblivious Turing machines by circuit families of small breadth.

Theorem 8.3.11. *Let M be an input-oblivious deterministic multi-tape Turing machine such that on inputs of length n, M runs in time $t(n)$ and space $s(n) \geq \log n$. Then there exists a family of circuits simulating M such that each circuit has size $O(t(n)\log s(n))$, depth $t(n)$, and breadth $O(s(n))$.*

Proof. As in the proof of Lemma 8.3.6, we shall only describe the construction of Theorem 8.3.9 by Schnorr [124]. See Schnorr [124] and Wegener [146] for the proof for the size bound. The bounds for depth and breadth will be clear from the construction below.

Let M be the given input-oblivious Turing machine, and let M' be the oblivious Turing machine in Theorem 8.3.3. By Lemma 8.3.7, the first three channels on

the first work tape of M' contain the input, the witnessing Turing machine of $t(n)$ (the yardstick), and the output. The output has 1 bit for decision problems, and may have $t(n)$ bits otherwise. In the case where the output has more than 1 bit, we simply construct a subcircuit each time M writes a symbol to the output tape, such that the subcircuit outputs the same value as the symbol. Note that this will give a multiple-output circuit. The space used on the first two channels are n and $t(n)$. The second work tape of M' contains a counter and a temporary space for the copy/split/combine operations in the $clean(m)$ subprogram, and the space used on it is at most $O(\log s(n) + s(n))$.

Notice that, since our goal is to build a circuit simulating M' on inputs of length n, we do not need the witnessing Turing machine to compute the string of length $t(n)$. Furthermore, since M is oblivious on its input tape, the input bit used in each step of M is predetermined. So when applying the table method, instead of using the whole input of length n as in the classic table method, we only need to insert one particular input bit into each row of the table. Then for the first work tape of M', we only need to consider channels other than the first three, and these channels use spaces at most $O(s(n))$.

Now consider the second work tape of M'. Recall that this work tape is used in the $clean(m)$ subprogram of M', and the space used is at most $O(s(n))$. Also recall that the $clean(m)$ subprogram consists of three operations: copy a single block, combine two blocks into one block, and split one block into two blocks. All these operations can be easily implemented by a constant-depth circuit with multiple outputs, since the copy/split/combine operations of channels in a block can be done in parallel. See Figure 8.3.

Now consider the subcircuits corresponding to the $clean(m)$ subprogram. Observe that between different subprograms, M' simply moves its tape heads to the target location without changing the work tapes. For example, in $clean(m)$, M' simply moves its head from the end of the m-th segment to the beginning of the m-th segment. So we can ignore subcircuits generated by those rows in the table method corresponding to these trivial head traversals. Also notice that, in the subprogram $clean(m)$, after M' cleans the segment $S_{c,m}$, M' will go to the segment $S_{c,-m}$. We can also remove those rows corresponding to this head movement. Furthermore, since cleaning the two segments $S_{c,m}$ and $S_{c,-m}$ are independent, the corresponding subcircuits can be put in parallel in the resulting circuit.

Figure 8.3: The subcircuits for $sim(0)$ and $clean(m)$ in the oblivious Turing machine

Our final modification is that, since M' uses $O(s(n))$ space on its work tapes excluding the first three channels on the first work tape, there are no more than $\lceil \log s(n) \rceil$ number of segments in a channel. Thus when constructing the circuit, we can simply ignore all calls of $clean(k)$ where $k > \log s(n)$. Also observe that the broadness of each layer is $O(s(n))$, so the final circuit will have breadth $O(s(n))$. □

See Figure 8.4 for an example of the construction. The input $x_1, \ldots, x_n$ are put inside of each $sim(0)$ subcircuit. All other inputs are constants indicating blank symbols on the work tapes. For simplicity, we group wires between the same cells together as one wire. Since the number of wires between the same cells is a constant, each wire in Figure 8.4 represents a constant number of wires in the actual circuit.

Using Theorem 8.3.3, Cook [26] proved the following result.

Theorem 8.3.12. *[26] Let N be a non-deterministic multi-tape Turing machine running in time $O(t(n))$ on inputs of length n. Then for each input x, there exists a circuit C of size $O(t(n) \log t(n))$ such that C on x is satisfiable iff N accepts x.*

Proof. (Sketch) First convert N into a deterministic Turing machine M by an auxiliary input tape, which stores the non-deterministic choices N will make during the computation. Using the construction of Theorem 8.3.3, we then have a circuit family such that each circuit C accepts (x, c) iff M accepts (x, c), where x is the input to N of length n, and c is the string of length $t(n)$ representing the choices made by N during the computation. The claim then follows. □

Corollary 8.3.13. *Let N be an input-oblivious non-deterministic multi-tape Turing machine such that on inputs of length n, N runs in time $t(n)$ and space $s(n)$. Then there exists a family of circuits such that each circuit C has size $O(t(n) \log(n + s(n) + t(n)))$, depth $t(n)$, breadth $O(s(n))$, and C is satisfiable iff N accepts input x.*

Proof. Note that the deterministic machine in Cook's proof for Theorem 8.3.12 is oblivious on its auxiliary tape. The claim then follows from Theorem 8.3.11 and Theorem 8.3.12. □

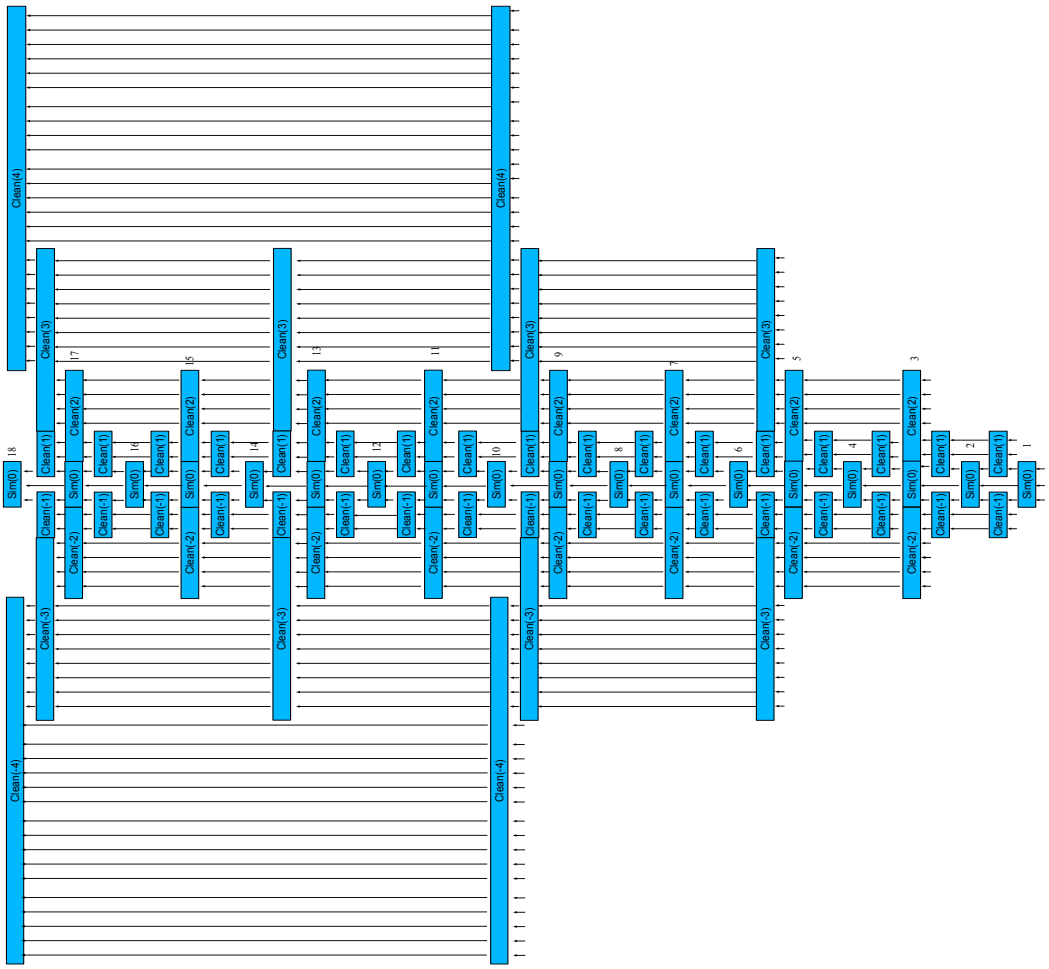

Figure 8.4: An overview of a sample circuit from the oblivious Turing machine. The number to the right of each $sim(0)$ subcircuit denotes the computation step of the oblivious Turing machine simulated by the subcircuit.

8.3.3 Simultaneous Time and Space Product of Input-Oblivious Turing Machines, Planar Circuits, and Circuit Depth

In this section, we give two theorems relating simultaneous time and space product of input-oblivious Turing machines first to planar circuit size, then to circuit depth.

Theorem 8.3.14. *Let M be an input-oblivious deterministic multi-tape Turing machine running in time $t(n)$ and space $s(n)$ on inputs of length n. Then there exists a family of planar Boolean circuits of size $O(t(n)s(n))$ simulating M.*

Proof. Let C be the Boolean circuit constructed from M in Theorem 8.3.11. We want to count the number of crossings in C. By Theorem 8.1.15, it is easy to show that the number of crossings is $O(t(n)s(n)\log s(n))$, since C has size $O(t(n)\log s(n))$ and breadth $O(s(n))$. The following proof further reduces the number of crossings to $O(t(n)s(n))$.

First notice that each subcircuit corresponding to the $sim(0)$ subprogram has constant number of crossings, since these subcircuits have constant size. So there are $O(t(n))$ number of crossings in those subcircuits for all the $sim(0)$ subprograms.

Let k be a non-negative integer. Now we count the number of crossings in the subcircuits for the $clean(k)$ subprograms. As mentioned in the proof for Theorem 8.3.9 and Theorem 8.3.11, we only need to consider the case when $k \leq \log s(n)$. The $clean(k)$ subprograms cleans the kth segment, so it manipulates data in the kth and $(k+1)$th segments, which have $2^{k-1} + 2^k = \theta(2^k)$ cells in total. So the number of crossings in the corresponding subcircuit for $clean(k)$ is

$$2\binom{\theta(2^k)}{2} = \theta(2^{2k}).$$

(It is multiplied by 2 since $clean(k)$ also cleans the $(-k)$th segment.) By Lemma 8.3.8, the subcircuit for $clean(k)$ occurs $t(n)/2^{k-1}$ times. So the number of crossings in all the $clean(k)$'s subcircuit is

$$\theta\left(\frac{t(n)}{2^{k-1}}2^{2k}\right) = \theta\left(t(n)2^k\right).$$

Then the total number of crossings corresponding to all the *clean* subprograms is

$$\theta\left(\sum_{k=0}^{\log s(n)} t(n)2^k\right) = \theta(t(n)s(n)).$$

Thus the number of crossings in C is $\theta(t(n)s(n))$. By Lemma 8.1.14, we can construct a family of planar circuits simulating M with size $O(t(n)s(n))$. □

Theorem 8.3.15. *Let M be an input-oblivious deterministic multi-tape Turing machine running in time $t(n)$ and space $s(n)$ on inputs of length n. Then there exists a family of circuits of depth $O(\sqrt{t(n)s(n)})$ simulating M.*

Proof. By Theorem 8.3.14, there exists a planar Boolean circuit C with size $O(t(n)s(n))$ equivalent to M on inputs of length n. By our generalization of Spira's theorem (Theorem 4.1.1) and Lipton and Tarjan's separator theorem for planar graphs [84], we can simulate C by a Boolean circuit of depth $O(\sqrt{t(n)s(n)})$. □

Recall that in Theorem 1.2.1, Borodin [11] showed that languages decided in simultaneous time $t(n)$ and space $s(n)$ can be decided by circuit families with depth $O(s(n)\log t(n))$. Note that in the case of input-oblivious Turing machines, Theorem 8.3.15 gives a better simulation of simultaneous time and space by depth when $s(n) > t(n)/\log^2 t(n)$, since in this case we have $\sqrt{s(n)t(n)} \leq s(n)\log t(n)$.

Theorem 8.3.16. *Let N be an input-oblivious non-deterministic multi-tape Turing machine running in time $t(n)$ and space $s(n)$ on inputs of length n. Then there exists a family of circuits of depth $O(\sqrt{t(n)s(n)})$ such that for each circuit C, C on x is satisfiable iff N accepts x.*

Proof. By Corollary 8.3.13, there exists a family of circuits such that each circuit C has size $O(t(n)\log s(n))$, depth $t(n)$, breadth $O(s(n))$, and C is satisfiable iff N accepts input x. To prove the claim, simply apply the same arguments in the proofs for Theorem 8.3.14 and Theorem 8.3.15 to this circuit family. □

8.3.4 Quadratic Lower Bounds for the Simultaneous Time and Space Product of Input-Oblivious Turing Machines

We first review some known results about lower bounds on time and space product. Let n be the input length. On deterministic multi-tape Turing machines, Cobham

[22] gave $\Omega(n^2)$ lower bounds on time and space product for palindromes and related languages. One can also use communication complexity arguments to show quadratic lower bounds on time and space product of deterministic multi-tape Turing machines. See Kushilevitz and Nisan [76]. As far as we know, these are the only techniques that directly give $\Omega(n^2)$ on the time and space product of Turing machines.

Branching programs were first defined by Masek [93]. Also see Razborov [114] for a comprehensive survey. The *time* of a branching program is the maximum length over all paths in the branching program, and the *space* is the base-2 logarithm of the number of nodes. It can be easily shown that if a language L is decided by a deterministic multi-tape Turing machine in time $t(n)$ and space $s(n) \geq \log n$, then there exists a family of branching programs for L that runs in time $O(t(n))$ and in space $O(s(n))$. The same bounds are also true if L is decided by a RAM running in unit-cost time $t(n)$ and log-cost space $s(n)$. Thus for time and space product, lower bounds on branching programs can be translated into lower bounds on deterministic multi-tape Turing machines and RAMs. This motivates the study of time and space product on branching programs. See the survey by Borodin [12] and Chapter 10 by Savage [121] for results in this direction. Here we mention two relevant results. For matrix multiplication of two $m \times m$ matrices over the integers or over finite prime fields, Yesha [152] gave a $\Omega(n^{1.5})$ bound on the time and space product on branching programs, where $n = \Omega(m^2)$ is the input length. Note that in the case of finite prime fields, this is tight on (unit-cost time and log-cost space) RAMs since the time and space product takes $O(n^{1.5})$ using the definition of matrix multiplication. Mansour, Nisan, and Tiwari [90] also proved that any implementation of a universal hashing function from n-bit strings to m-bit strings (e.g. $x + y \cdot z$ where x, y, z are from a field F of size n) requires $\Omega(mn)$ time and space product on branching programs.

Lower bounds on time and space product in other models were also studied. A series of results [67, 39, 82, 41, 133, 40, 31, 148, 147, 149, 18] studied the lower bounds on time and space product for SAT. The current best result is by Williams [147, 149], showing that if SAT is decided by the machine running in time n^c and space n^ϵ, then $c + \epsilon \geq 2\cos(\pi/7) \approx 1.8019....$ The machines considered can be deterministic and non-deterministic Turing machines, as well as unit-cost and log-cost RAMs. Santhanam [120] gave a lower bound $\Omega(n^2/poly\log(n))$ on the time and space product of SAT on non-deterministic Turing machines.

In this section, we show $\Omega(n^2)$ lower bounds for the product of deterministic simultaneous time and space of input-oblivious Turing machines for certain multiple-output and single-output Boolean functions including matrix multiplications. In the case of matrix multiplications, our lower bound is tight.

We obtain these results by a new approach. Our argument uses oblivious Turing machines as a tool. As noted above, time and space product lower bounds on branching programs imply time and space product lower bounds on Turing machines, as well as RAMs (unit-cost time and log-cost space). In the case of matrix multiplication over finite fields, the $\Omega(n^{1.5})$ bound is tight on input-oblivious RAMs. Thus, in the case of matrix multiplication our approach gives a stronger bound than what is achievable by branching programs. Also note that a super-quadratic lower bound on time and space product for computing some explicit function would imply a super-linear lower bound on time of Turing machines computing it.

Definition 8.3.17. Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ be two Boolean vectors. The *Boolean convolution* of x and y is defined to be $z = (z_2, \ldots, z_{2n})$, where

$$z_k = \bigvee_{i+j=k} x_i y_j.$$

In [83], Lipton and Tarjan showed the following three lower bounds on planar circuit size.

Theorem 8.3.18. *[83]*

1. *Any planar Boolean circuit computing the Boolean convolution of two vectors in $\{0,1\}^n$ has at least $\Omega(n^2)$ gates.*

2. *Any planar Boolean circuit computing the product of two n-bit binary integers has at least $\Omega(n^2)$ gates.*

3. *Any planar Boolean circuit computing the matrix product of two $m \times m$ Boolean matrix has at least $\Omega(n^2)$ gates, where $n = O(m^2)$ is the size of the input.*

Theorem 8.3.18 implies $\Omega(n^2)$ lower bounds on time and space product of input-oblivious Turing machines for all three functions. These bounds on Boolean convolution and integer multiplication already follow from the results by Savage and

Swamy [122] on straight-line programs. Here we state our lower bound result for Boolean matrix multiplication. As far as we know, this bound on matrix multiplication does not follow from earlier results.

Theorem 8.3.19. *For any input-oblivious deterministic multi-tape Turing machine, computing the matrix multiplication of two Boolean $m \times m$ matrices in time $t(n)$ and space $s(n)$ must satisfy $t(n)s(n) = \Omega(n^2)$, where $n = \theta(m^2)$ is the input length.*

Proof. By Theorem 8.3.14, there exists a planar circuit of size $O(t(n)s(n))$ computing matrix multiplication. By Theorem 8.3.18, we have $t(n)s(n) = \Omega(n^2)$. □

It is not hard to see that our bound for matrix multiplication is tight.

Next we consider lower bounds for single-output Boolean functions. In [135], Turán showed the following lower bound.

Theorem 8.3.20. *[135] There exists a single-output Boolean function f such that any planar circuit computing f with n inputs has at least $\Omega(n^2)$ gates.*

Turán gave an explicit construction for the function. Here is our lower bound result for single output functions.

Theorem 8.3.21. *Let f be the Boolean function defined in Theorem 8.3.20. Let $t(n)$ and $s(n)$ be the time and space complexity of any input-oblivious multi-tape Turing machine computing f with input length n. Then we have $t(n)s(n) = \Omega(n^2)$.*

Proof. Same as the proof for Theorem 8.3.19. □

Bibliography

[1] Anna Adamaszek and Michal Adamaszek. Combinatorics of the change-making problem. *European Journal of Combinatorics*, 31(1):47–63, 2010. 76

[2] Leonard M. Adleman and Michael C. Loui. Space-bounded simulation of multitape turing machines. *Theory of Computing Systems*, 14(1):215–222, 1981. 5

[3] Eric Allender. Arithmetic circuits and counting complexity classes. In *Complexity of Computations and Proofs*, volume 13, pages 33–72, 2004. 21

[4] Eric Allender, Jia Jiao, Meena Mahajan, and V. Vinay. Non-commutative arithmetic circuits: Depth reduction and size lower bounds. *Theor. Comput. Sci*, 209(1-2):47–86, 1998. 22, 23, 65

[5] Noga Alon, Paul Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and its applications. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 293–299, Baltimore, Maryland, 14–16 May 1990. 20, 80

[6] D. A. M. Barrington, C.-J. Lu, P. B. Miltersen, and S. Skyum. On monotone planar circuits. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity (CCC-99)*, pages 24–33, Los Alamitos, May 4–6 1999. IEEE Computer Society. 25

[7] Cristina Bazgan, Miklos Santha, and Zsolt Tuza. Efficient approximation algorithms for the SUBSET-SUMS EQUALITY problem. *J. Comput. Syst. Sci*, 64(2):160–170, 2002. 72

[8] Edward G. Belaga. Locally synchronous complexity in the light of the transbox method. In *Symposium of Theoretical Aspects of Computer Science*, volume 166 of *lncs*, pages 129–139, Paris, France, 11–13 April 1984. Springer. 29, 31

[9] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957. 71, 73

[10] Maria Luisa Bonet and Samuel R. Buss. Size-depth tradeoffs for Boolean formulae. *Information Processing Letters*, 49(3):151–155, 11 February 1994. 4, 18

[11] Allan Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, December 1977. 3, 5, 26, 58, 64, 65, 95, 114

[12] Allan Borodin. Time space tradeoffs (getting closer to the barrier?). In *ISAAC*, volume 762 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 1993. 115

[13] Mark Braverman, Stephen A. Cook, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Branching programs for tree evaluation. In *MFCS*, volume 5734, pages 175–186. Springer, 2009. 17

[14] Mark Braverman, Stephen A. Cook, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Fractional pebbling and thrifty branching programs. In *FSTTCS*, volume 4, pages 109–120. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009. 17

[15] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974. 5, 21, 22

[16] Nader H. Bshouty, Richard Cleve, and Wayne Eberly. Size-depth tradeoffs for algebraic formulas. *SIAM Journal on Computing*, 24(4):682–705, August 1995. 5

[17] S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, August 1992. 8, 24, 25

[18] Sam Buss and Ryan Williams. Limits on alternation-trading proofs for time-space lower bounds. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:31, 2011. 115

[19] Samuel R. Buss. The boolean formula value problem is in alogtime. In *in Proceedings of the 19-th Annual ACM Symposium on Theory of Computing*, pages 123–131, 1987. 8, 25

[20] T. Chakraborty and S. Datta. One-input-face mpcvp is hard for l, but in logdcfl. *Proceedings of FSTTCS 2006, LNCS 4337*, pages 57–68, 2006. 26

[21] Siu Man Chan. Just a pebble game. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:42, 2013. 24

[22] Alan Cobham. The recognition problem for the set of perfect squares. In *Conference Record of 1966 Seventh Annual Symposium on Switching and Automata Theory*, pages 78–87, Berkeley, California, 26–28 October 1966. IEEE. 115

[23] Anne Condon. A theory of strict P-completeness. *Computational Complexity*, 4(3):220–241, 1994. 26

[24] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 3–5 1971 1971. 12, 93

[25] Stephen A. Cook. An observation on time-storage trade off. *Journal of Computer and System Sciences*, 9(3):308–316, December 1974. 15, 17, 33, 42, 100

[26] Stephen A. Cook. Short propositional formulas represent nondeterministic computations. *Information Processing Letters*, 26(5):269–270, 11 January 1988. Note. 111

[27] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *CoRR*, abs/1005.2642, 2010. informal publication. 17

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. 71, 72

[29] George Dantzig. Discrete-variable extremum problems. *Operations Research*, 5:266–277, 1957. 71, 73

[30] Arthur L. Delcher and S. Rao Kosaraju. An NC algorithm for evaluating monotone planar circuits. *SIAM J. Comput*, 24(2):369–375, 1995. 26

[31] Scott Diehl and Dieter van Melkebeek. Time-space lower bounds for the polynomial-time hierarchy on randomized machines. *SIAM Journal on Computing*, 36(3):563–594, 2006. 115

[32] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999. 47

[33] Patrick W. Dymond and Stephen A. Cook. Complexity theory of parallel time and hardware. *Information and Computation*, 80(3):205–226, 1989. 25

[34] Patrick W. Dymond and Martin Tompa. Speedups of deterministic machines by synchronous parallel machines. *Journal of Computer and System Sciences*, 30(2):149–161, April 1985. vi, 4, 6, 7, 9, 10, 15, 16, 17, 18, 28, 37, 100

[35] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *Proceedings of FOCS*, pages 143–152. IEEE Computer Society, 2010. 73, 74, 75, 81, 82

[36] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In *Proceedings of STACS*, volume 14 of *LIPIcs*, pages 66–77, 2012. 73

[37] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Real-time simulation of multihead tape units. *Journal of the ACM*, 19(4):590–607, October 1972. 102

[38] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006. 47

[39] Lance Fortnow. Nondeterministic polynomial time versus nondeterministic logarithmic space: Time-space tradeoffs for satisfiability. In *Proceedings, Twelfth Annual IEEE Conference on Computational Complexity*, pages 52–60, Ulm, Germany, 24–27 June 1997. IEEE Computer Society Press. 115

[40] Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *Journal of the ACM*, 52(6):835–865, November 2005. 115

[41] Lance Fortnow and Dieter van Melkebeek. Time-space tradeoffs for nondeterministic computation. In *IEEE Conference on Computational Complexity*, pages 2–13. IEEE Computer Society, 2000. 115

[42] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, California, 1–3 May 1978. 6

[43] A. Gál, M. Koucky, and P. McKenzie. Incremental branching programs. *Theory of Computing Systems*, 43(2):159–184, 2008. 17

[44] Anna Gál and Jing-Tang Jang. The size and depth of layered boolean circuits. In *Proceedings of LATIN*, volume 6034 of *Lecture Notes in Computer Science*, pages 372–383. Springer, 2010. 29

[45] Anna Gál and Jing-Tang Jang. The size and depth of layered boolean circuits. *Information Processing Letters*, 111(5):213–217, 2011. 29

[46] Anna Gál and Jing-Tang Jang. A generalization of spira's theorem and circuits with small segregators or separators. In *SOFSEM 2012*, volume 7147 of *Lecture Notes in Computer Science*, pages 264–276. Springer, 2012. See full version in *A Generalization of Spira's Theorem and Circuits with Small Segregators or Separators, Electronic Colloquium on Computational Complexity (ECCC), 20:93, 2013.* 44

[47] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman, New York, 1979. 71, 75

[48] Gaskov. The depth of boolean functions. *Probl. Kibernet.*, 34:265–268, 1978. 4, 36

[49] Gens and Levner. Complexity of approximation algorithms for combinatorial problems: A survey. *SIGACTN: SIGACT News*, 12, 1980. 72

[50] G. V. Gens and E. V. Levner. Computational complexity of approximation algorithms for combinatorial problems. In *Mathematical Foundations of Computer Science 1979*, volume 74 of *lncs*, pages 292–300. Springer-Verlag, 1979. 72

[51] John R. Gilbert, Joan P. Hutchinson, and Robert Endre Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5(3):391–407, September 1984. 20

[52] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM Journal on Computing*, 9(3):513–524, August 1980. 24

[53] L. Goldschlager. The monotone and planar circuit value problem is complete for P. *SIGACT News*, pages 25–27, 1977. vii, 8, 11, 25, 55

[54] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, Oxford-New York, 1995. vii, 11, 25, 55

[55] Xaver Gubáš, Juraj Hromkovič, and Juraj Waczulík. A nonlinear lower bound on the practical combinational complexity. *Theoretical Computer Science*, 143(2):335–342, 12 June 1995. Note. 9

[56] Hansen, Miltersen, and Vinay. Circuits on cylinders. *CMPCMPL: Computational Complexity*, 15:62–81, 2006. 42

[57] L.H. Harper. An $n \log n$ lower bound on synchronous combinational complexity. *Proc. AMS*, 64(2):300–306, 1977. 29

[58] Philipp Hertel and Toniann Pitassi. The PSPACE-completeness of black-white pebbling. *SIAM Journal on Computing*, 39(6):2622–2682, 2010. 24

[59] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *Journal of the ACM*, 24(2):332–337, April 1977. 5, 14, 15, 100

[60] Juraj Hromkovič. Nonlinear lower bounds on the number of processors of circuits with sublinear separators. *Information and Computation*, 95(2):117–128, December 1991. 9

[61] Laurent Hyafil and Ronald L. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Technical Report Rapport de Recherche no. 33, IRIA – Laboratoire de Recherche en Informatique et Automatique, October 1973. 84, 100

[62] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, October 1975. 72

[63] Maurice Jansen and Jayalal M. N. Sarma. Balancing bounded treewidth circuits. In Farid M. Ablayev and Ernst W. Mayr, editors, *CSR*, volume 6072 of *Lecture Notes in Computer Science*, pages 228–239. Springer, 2010. vii, 47

[64] Camille Jordan. Sur les assemblages des lignes. *Journal für die reine und angewandte Mathematik*, 70:185–190, 1869. 18, 20

[65] Bala Kalyanasundaram and Georg Schnitger. Rounds versus time for the two person pebble game. *Information and Computation*, 88(1):1–17, September 1990. 16, 17

[66] Daniel M. Kane. Unary subset-sum is in logspace. *CoRR*, abs/1012.1336, 2010. 73, 74, 75

[67] Ravindran Kannan. Towards separating nondeterminism from determinism. *Mathematical Systems Theory*, 17(1):29–45, April 1984. 115

[68] Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require super-logarithmic depth. *SIAM Journal on Discrete Mathematics*, 3(2):255–265, May 1990. 24

[69] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computation*, pages 85–103, 1972. 12, 72, 75

[70] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 17, pages 870–941. North Holland, 1990. 7, 25

[71] Hans Kellerer, Renata Mansini, Ulrich Pferschy, and Maria Grazia Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *J. Comput. Syst. Sci*, 66(2):349–370, 2003. 72

[72] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004. 76

[73] S. Rao Kosaraju. On the parallel evaluation of classes of circuits. In *Foundations of Software Technology and Theoretical Computer Science*, pages 232–237. Springer, 1990. 25

[74] Dexter Kozen and Shmuel Zaks. Optimal bounds for the change-making problem. *Theoretical Computer Science*, 123(2):377–388, 31 January 1994. 76

[75] Matthias P. Krieger. On the incompressibility of monotone DNFs. In Maciej Liskiewicz and Rüdiger Reischuk, editors, *FCT*, volume 3623 of *Lecture Notes in Computer Science*, pages 32–43. Springer, 2005. 65

[76] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997. 24, 115

[77] R. E. Ladner. The circuit value problem is log-space complete for P. *SIGACT News*, 6(2):18–20, 1975. 8, 25

[78] Thomas Lengauer. Black-white pebbles and graph separation. *Acta Informatica*, 16:465–475, 1981. 32

[79] N. Limaye, M. Mahajan, and J. Sarma. Upper bounds for monotone planar circuit value and variants. *Computational Complexity*, 18:377–412, 2009. 26

[80] Nutan Limaye, Meena Mahajan, and Jayalal M. N. Sarma. Evaluating monotone circuits on cylinders, planes and tori. In *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, volume 3884 of *Lecture Notes in Computer Science*, pages 660–671. Springer, 2006. 42

[81] Nutan Limaye, Meena Mahajan, and Karteek Sreenivasaiah. The complexity of unary subset sum. In *COCOON*, volume 7434 of *Lecture Notes in Computer Science*, pages 458–469. Springer, 2012. 73, 74, 76, 77, 78, 79, 82, 84

[82] R. J. Lipton and A. Viglas. On the complexity of SAT. In *40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 459–464. IEEE, 1999. 115

[83] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, August 1980. 29, 37, 80, 116

[84] Lipton, R. J. and Tarjan, R. E. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979. 9, 12, 20, 80, 81, 114

[85] Daniel Lokshtanov and Jesper Nederlof. Saving space by algebraization. In ACM, editor, *STOC'10: Proceedings of the 2010 ACM International Symposium on Theory of Computing: June 5–8, 2010, Cambridge, MA, USA*, pages 321–330, pub-ACM:adr, 2010. 73, 75

[86] George Lueker. Two np-complete problems in nonnegative integer programming. Technical Report 178, Computer Science Laboratory, Princeton University, 1975. 76

[87] N. A. Lynch. Log space recognition and translation of parenthesis languages. *J. Assoc. Comput. Mach.*, 24:583–590, 1977. 8, 25

[88] Meena Mahajan and B. V. Raghavendra Rao. Small-space analogues of valiant's classes. In *Proceedings of 17th Fundamentals of Computation Theory (FCT)*, volume 5699 of *Lecture Notes in Computer Science*, pages 250–261. Springer, 2009. 64

[89] Guillaume Malod and Natacha Portier. Characterizing valiant's algebraic complexity classes. *J. Complexity*, 24(1):16–38, 2008. 63

[90] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 235–243, Baltimore, Maryland, 14–16 May 1990. 115

[91] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990. 76

[92] William Joseph Masek. A fast algorithm for the string editing problem and decision graph complexity. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1976. 63

[93] William Joseph Masek. A fast algorithm for the string editing problem and decision graph complexity. Master's thesis, 1976. 115

[94] Ernst W. Mayr and Ashok Subramanian. The complexity of circuit value and network stability. *Journal of Computer and System Sciences*, 44(2):302–323, April 1992. 25

[95] W.F. McColl and M.S. Paterson. The depth of all boolean functions. *SIAM J. on Comp.*, 6:373–380, 1977. 4, 13, 36

[96] McKenzie, Reinhardt, and Vinay. Circuits and context-free languages. In *COCOON: Annual International Conference on Computing and Combinatorics*, pages 194–203, 1999. 64

[97] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, June 1986. 80

[98] Gary L. Miller, Vijaya Ramachandran, and Erich Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 17(4):687–695, August 1988. 22

[99] Nisan and Wigderson. Lower bounds on arithmetic circuits via partial derivatives. *Computational Complexity*, 6(3):217–234, 1997. 23

[100] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. 1, 5, 6, 93, 108

[101] M. S. Paterson and L. G. Valiant. Circuit size is nonlinear in depth. *Theoretical Computer Science*, 2(3):397–400, September 1976. vi, 4, 9, 10, 13, 16, 17, 28

[102] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. *Mathematical Systems Theory*, 10:239–251, 1977. 15, 29, 33, 37, 100

[103] Pippenger and Fischer. Relations among complexity measures. *JACM: Journal of the ACM*, 26:361–381, 1979. 3, 5, 6, 12, 86, 93, 101, 102, 103, 107

[104] N. Pippenger. Pebbling. In *Proceedings of the Fifth IBM Symposium on Mathematical Foundations of Computer Science*, 1980. 14, 17

[105] N. Pippenger. Advances in pebbling. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 1982. 14

[106] Nicholas Pippenger. On simultaneous resource bounds. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 307–311, 1979. 87

[107] Nicholas Pippenger. Comparative schematology and pebbling with auxiliary pushdowns (preliminary version). In *Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 351–356, Los Angeles, California, 28–30 April 1980. 17

[108] Ashok Kumar Ponnuswami and H. Venkateswaran. Monotone multilinear boolean circuits for bipartite perfect matching require exponential size. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2004. 65

[109] Vijaya Ramachandran and Honghua Yang. An efficient parallel algorithm for the general planar monotone circuit value problem. *SIAM Journal on Computing*, 25(2):312–339, 1996. 26

[110] Vijaya Ramachandran and Honghua Yang. An efficient parallel algorithm for the layered planar monotone circuit value problem. *Algorithmica*, 18(3):384–404, 1997. 26

[111] Ran Raz. Multi-linear formulas for permanent and determinant are of super-polynomial size. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 633–641, New York, June 13–15 2004. ACM Press. 23, 65

[112] Ran Raz. Separation of multilinear circuit and formula size. In *Theory of Computing – An Open Access Journal*, volume 2. 2006. 65

[113] Ran Raz and Amir Yehudayoff. Balancing syntactically multilinear arithmetic circuits. *Journal of Computational Complexity*, 17(4):515–535, 2008. 23, 65

[114] A. A. Razborov. Lower bounds for deterministic and nondeterministic branching programs. In *Proceedings of Fundamentals of Computation Theory (FCT)*, volume 529 of *LNCS*, pages 47–60. Springer, 1991. 63, 115

[115] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), 2008. 82, 83

[116] N. Robertson and P. D. Seymour. Graph Minors II. algorithmic aspects of tree width. *Journal of Algorithms*, 7:309–322, 1986. 47

[117] Arnold L. Rosenberg and Lenwood S. Heath. *Graph Separators, with Applications*. Kluwer Academic, 2001. 9, 100

[118] W. L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981, June. 6

[119] Rahul Santhanam. On separators, segregators and time versus space. In Frances M. Titsworth, editor, *Proceedings of the Sithteenth Annual Conference on Computational Complexity (CCC-01)*, pages 286–294, Los Alamitos, CA, June 18–21 2000. IEEE Computer Society. 19

[120] Rahul Santhanam. Lower bounds on the complexity of recognizing SAT by Turing machines. *Information Processing Letters*, 79(5):243–247, 2001. 115

[121] John E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998. 115

[122] John E. Savage and Sowmitri Swamy. Space-time tradeoffs for oblivious integer multiplications. In *Automata, Languages and Programming, 6th Colloquium*, volume 71 of *Lecture Notes in Computer Science*, pages 498–504. Springer-Verlag, 16–20 July 1979. 117

[123] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970. 49, 57

[124] Claus-Peter Schnorr. The network complexity and the Turing machine complexity of finite functions. *Acta Informatica*, 7:95–107, 1976. 12, 93, 108

[125] Rimli Sengupta and H. Venkateswaran. Multilinearity can be exponentially restrictive (preliminary version). Technical Report GIT-CC-94-40, Georgia Institute of Technology. College of Computing. 65

[126] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, 1975. 32

[127] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010. 21

[128] S. Skyum and L. G. Valiant. A complexity theory based on boolean algebra. *Journal of the ACM*, 32(2):484–502, 1985. 11, 60

[129] P.M. Spira. On time-hardware complexity tradeoffs for boolean functions. *In Proc. 4th Hawaii Symp. on System Sciences*, pages 525–527, 1971. vi, 4, 10, 18, 41, 44

[130] Larry Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13(2):409–422, May 1984. 4, 7

[131] Seinosuke Toda. Classes of arithmetic circuits capturing the complexity of computing the determinant. *IEICE Transactions on Communications/Electronics/Information and Systems*, pages 116–124, 1992. 63

[132] Martin Tompa. A pebble game that models alternation. *Unpublished manuscript.* 17, 31

[133] Iannis Tourlakis. Time-space tradeoffs for SAT on nonuniform machines. *J. Comput. Syst. Sci*, 63(2):268–287, 2001. 115

[134] G. Turán. On restricted boolean circuits. In *Fundamentals of Computation Theory (FCT '89)*, pages 460–469, Berlin - Heidelberg - New York, August 1989. Springer. 28, 31

[135] György Turán. Lower bounds for synchronous circuits and planar circuits. *Information Processing Letters*, 30(1):37–40, 16 January 1989. 117

[136] Jeffrey D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Maryland, 1984. 37, 80

[137] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal on Computing*, 12(4):641–644, November 1983. 11, 22, 23, 60, 62, 65, 66, 68, 69

[138] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001. 72

[139] H. Venkateswaran. Properties that characterize LOGCFL. *Journal of Computer and System Sciences*, 43(2):380–404, October 1991. 64

[140] H. Venkateswaran. Circuit definitions of nondeterministic complexity classes. *SIAM Journal on Computing*, 21(4):655–670, August 1992. 63

[141] H. Venkateswaran and Martin Tompa. A new pebble game that characterizes parallel complexity classes. *SIAM Journal on Computing*, 18(3):533–549, June 1989. 37, 100

[142] Jeffrey Scott Vitter and Roger A. Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Trans. Computers*, 35(5):403–418, 1986. 25, 26

[143] Heribert Vollmer. *Introduction to circuit complexity*. Springer-Verlag, 1999. 26, 102, 103, 107

[144] E. Szemerédi W. J. Paul, N. Pippenger and W. T. Trotter. On determinism versus non-determinism and related problems. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 429–438, 1983. 19

[145] Ingo Wegener. Relating monotone formula size and monotone depth of Boolean functions. *Information Processing Letters*, 16(1):41–42, 24 January 1983. 4, 10, 18, 44, 47

[146] Ingo Wegener. *The Complexity of Boolean Functions*. B. G. Teubner, and John Wiley & Sons, 1987. 2, 9, 36, 61, 92, 102, 103, 108

[147] R. Ryan Williams. Time-space tradeoffs for counting NP solutions modulo integers. *Computational Complexity*, 17(2):179–219, 2008. 115

[148] Ryan Williams. Inductive time-space lower bounds for sat and related problems. *Computational Complexity*, 15(4):433–470, 2006. 115

[149] Ryan Williams. Alternation-trading proofs, linear programming, and lower bounds. In *STACS*, volume 5, pages 669–680. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. 115

[150] Gerhard J. Woeginger and Zhongliang Yu. On the equal-subset-sum problem. *Information Processing Letters*, 42(6):299–302, 1992. 72

[151] Honghua Yang. An NC algorithm for the general planar monotone circuit value problem. In *SPDP*, pages 196–203. IEEE Computer Society, 1991. 26

[152] Yaacov Yesha. Time-space tradeoffs for matrix multiplication and the discrete fourier transform of any general sequential random-access computer. *Journal of Computer and System Sciences*, 29(2):183–197, October 1984. 115

Vita

Jing-Tang Keith Jang earned a Bachelor of Science degree in Mathematics and a Bachelor of Science degree in Computer Science from Iowa State University in May 2004. In August 2004 he joined the Graduate School of the University of Texas, where he earned a Master of Science degree in Computer Science in December 2007.

Permanent Address: keithjtjang@gmail.com

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.